Versatile Communication Cost Modelling for Multicomputer Task Scheduling Heuristics

Maria Cristina Silva Boeres

Doctor of Philosophy
University of Edinburgh
1996

Para os Boeres

e para Vinod

Abstract

Programmers face daunting problems when attempting to design portable programs for multicomputers. This is mainly due to the huge variation in communication performance on the range of multicomputer platforms currently in use. These programmers require a computational model that is sufficiently abstract to allow them to ignore machine-specific performance features, and yet is sufficiently versatile to allow the computational structure to be mapped efficiently to a wide range of multicomputer platforms.

This dissertation focusses on parallel computations that can be expressed as *task graphs*: tasks that must be scheduled on the multicomputer's processors. In the past, scheduling models have only considered the message delay as the predominant communication parameter. In the current generation of parallel machines, however, latency is negligible compared to the *CPU penalty* of communication-related activity associated with inter-processor communication. This CPU penalty cannot be modelled by a latency parameter because the CPU activity consumes time otherwise available for useful computation. In view of this, we consider a model in which the CPU penalty is significant and is associated with *communication events* that are incurred when applications execute in parallel.

In this dissertation a new multi-stage scheduling approach that takes into account these communication parameters is proposed. Initially, in the first stage, the input task graph is transformed into a new structure that can be scheduled with a smaller number of communication events. Task replication is incorporated to produce clusters of tasks. However, a different view of clusters is adopted. Tasks are clustered so that messages are bundled and consequently, the number of communication events is decreased. The communication event tasks are associated with the relationship between the clusters. More specifically, this stage comprises a family of scheduling heuristics that can be customised to classes of parallel machines, according to their communication performance characteristics, through parameterisation and by varying the order in which the heuristics are applied. A second stage is necessary, where the actual schedule on

the target machine is defined. The mechanisms implemented analyse carefully the clusters and their relationship so that communication costs are minimised and the degree of parallelism is exploited. Therefore, the aim of the proposed approach is to tackle the *min-max* problem, considering realistic architectural issues.

Acknowledgements

A big thank you to all my friends who shared in my trials: Isabel and Thomas for my family environment; Anna for her understanding; Fred and Rob, my "Brazilian cousins", and all the others who shared in many important moments of my Scottish adventure. Thanks also to George, for the fruitful discussions,and to Murray and Jane for their patience to take a look at my thesis.

To my Brazilian friends Claudia, Lucia, Daniela, Marines, Otton, Satoru and Ricardo who always knew how to send comforting messages. To Wamberto, for receiving me warmily in Edinburgh.

Thanks to Vinod for being a constant and "patient" special friend, who always knew the right questions to ask and guided me to find the right path professionally. Most of all, more than anybody else, he was always sure that I could do it!

Para os meus pais, José e Léa, e irmãos, Claudia, Miguel e Sônia, que mesmo de longe, souberam sempre me mandar as palavras certas nos momentos certos e sempre torceram muito, passando para mim a certeza que eu conseguiria o meu objetivo com sucesso. Agradeço muito a eles, que mesmo nos momentos difíceis, me mostraram como a positividade e Fé nos leva a conseguir executar as nossas metas. Também agradeço ao resto da minha família muito querida.

Thanks to the Department of Computer Science, University of Edinburgh, for providing all the "essentials" for this work. To Departamento de Ciência de Computação, Universidade Federal Fluminense, for their support throughout the time I spent studying in Edinburgh.

Finally, this work was funded by a research studentship from the Brazilian Government, Ministry of Education - CAPES, under grant number 1553/91-8, for which I should thank the Brazilian People.

Declaration

This thesis was composed by myself and the work reported herein is my own except where indicated.

Dr. Peter Thanisch was my supervisor.

In the process of writing the following papers [125, 124], the relevant issues were discussed with Mike Norman and Peter Thanisch.

The work in [18] which analyses the performance of list scheduling algorithms was based on a discussion with George Chochia.

I participated in the final version of [24], since I was already proposing scheduling techniques that consider the CLAUD model.

A summary of the work (entitled "Versatile Ordering of DAG Scheduling Heuristics") described in this thesis was presented in the Workshop on Scheduling in Parallel and Distributed Systems, which took place at C.I.R.M., Marseille, 10-14 June 1996.

Maria Cristina Silva Boeres

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The field of task scheduling for parallel computers is a wide area of research and many scheduling algorithms have been proposed in the light of different characteristics of the target architecture. The scheduling problem is NP-complete in its general form [55] and therefore, approximation algorithms are proposed in order to produce a "good" schedule of an application onto a network of processors, usually, with time to completion within a constant factor of the optimal. The variety of architectural and application characteristics influence the development and collection of scheduling algorithms which also incorporate different scheduling techniques.

In the light of the evolution of parallel machines, many characteristics were identified as the *bottleneck* of the system and when implementing scheduling strategies, some of these characteristics were tackled. A model where the assumptions considered in the scheduling approach are specified is always necessary for the development of useful scheduling techniques.

The important characteristics that should be considered in the parallel field are captured in the many models of parallel computation, the evolution of which represents the need to tackle different issues of parallel computing and to represent parallel architectures more realistically. Since the proposal of the PRAM as a model of parallel computation [115], researchers have attempted to specify a model that captures realistic features of the parallel machines and their programming environment. The PRAM model was the first attempt to represent parallel computation, however, few would argue that it is not a very realistic model since communication and synchronisation costs are ignored. The first models, PRAM and various extensions [3, 4, 50], were formulated before the present generation of distributed-memory machines. Arguably, these models are not realistic representations of the current generation of parallel computers. However, the importance of these models for the analysis of the complexity of parallel algorithms is widely recognised.

With the development of distributed memory machines, more realistic models were proposed where communication characteristics were represented. Only in the last decade have academics had wide access to massively parallel machines. During all this period of time, performance characteristics have changed the nature of massively parallel machines, e.g. as the year-on-year performance improvement in processors outstripped that of other system components, the performance of the computer was hindered by slow link connecting processors. Tackling this problem, designers developed faster links and networking advances have made networks of workstations possible. Furthermore, recent advances in routing mechanisms have lessened the problems caused by different topologies implemented in various distributed memory machines. However, other *bottlenecks* have been identified, as for example, the interface between processors and the communication network.

Abstracting the users from specific details of the parallel machines, general models of parallel computation have been proposed with the objective of defining a bridge between architectural and programming models [34, 129]. These models are supposed to be more realistic, but at the same time, details are suppressed in order that the model may be more easily manipulated.

The evolution of models of parallel computation can be similarly observed in the scheduling field. Scheduling techniques were devised in order to overcome problems when scheduling applications on parallel computers. However, when devising a scheduling strategy with the objective of producing a "good" accurate schedule, relevant details of the application and target architecture must be defined in the scheduling model. Many general models of parallel computation do not specify enough of the details which are necessary in order to schedule an application efficiently.

On the analysis of models and machines of parallel computations, important parameters are identified. One parameter that has been given much attention is the delay due to communication. Although developments in network technology have improved communication performance, the delay is still considered as a relevant parameter that must be considered when parallelising an application. However, there is another communication performance characteristic that cannot be neglected: an overhead incurred on the processors when sending or receiving messages, which is not part of the delay, but rather a cost in the form of a *penalty* incurred in the processors which are communicating. Many scheduling models add this cost to the delay, considering that in the target machines there are dedicated processors which perform communication while other processors, only execute tasks. However, this approach is unsatisfactory for use with networks of workstations (NoWs). NoWs are fast machines, but the interface between processors and interconnection network is associated with high sending and receiving overheads, so it can be misleading to treat these costs as part of the delay. Furthermore, although vendors of parallel machines emphasise the efficiency of their communication systems, when using message passing platforms, such as MPI [43], high overheads have also been identified as a *blocking factor* on the communicating processors [63].

In the presence of overheads, *communication events* are considered as *tasks* executed by the processors. Therefore, when scheduling application in the target model, the scheduling approach must manipulate such kinds of tasks. In this dissertation we propose a *multi-stage approach* that considers architectural characteristics such as overheads, delay and a limited number of processors. Intuitively, in order to produce a good schedule, a reasonable strategy is to maximise the degree of parallelism and at the same time minimise the communication costs. Given attention to scheduling heuristics proposed in the literature, it is difficult to identify a schedule approach that takes into account the realistic characteristics specified in our target model. Also, a solution of such *min-max* problem is not always an objective for many scheduling approaches, since there are heuristics which produce a locally-optimal solution and also, there exist heuristics with performance that is data dependent.

In an attempt to decrease the number of *communication events* which hinder the efficiency of the schedule, in this dissertation we adopt an approach in which small messages are bundled into a larger one and in which tasks are clustered through task replication. However, a cluster of tasks is defined in a different way: no communication is carried out during the execution of the tasks inside a cluster.

Having defined the clusters, the communication events among them are specified by a *flexible* set of rules. The scheduling strategy devised in this thesis can produce good schedules for various applications and architectural models. We shall see from the way in which the rules were implemented, allowing customisation of their ordering and parameterisation that good schedules can be provided for different models.

After defining the *superdag* structure, a mapping mechanism is proposed based on Brent's Principle [20] which exploits the degree of parallelism and minimises communication costs (overheads and delays). Then, a new approach, called the *remapping* algorithm, incorporates the notion of locality and messages are bundled whenever necessary, but in such way that an attempt to decrease the communication delay is carried out.

1.1 Outline of the Thesis

The rest of this thesis is organised as follows.

In Chapter 2 models of parallel computation are classified. Firstly, abstract models which define memory as global (shared) are discussed, starting from the "simplest" models (the idealised models) towards those that incorporate more realistic issues. Then, distributed memory models are described considering the relevance of parallel computers with distributed memory, these models include some more realistic aspects, although relevant issues still need to be captured. Finally, general models of parallel computation proposed to be bridges between architectural and programming models are discussed. With the objective to give an insight into the relevant communication characteristics of real parallel computers, some parallel machines are described. Special attention is given to the network of workstations due to the importance of these machines for the parallel community, since many researchers argue that these are the parallel machines of the future. From the models and machines discussed, the characteristics that must be taken into account when dealing with problems like scheduling, are determined.

In Chapter 3 the target application and architectural models tackled in this thesis are defined. Important definitions related to *communication events* and the schedule considering our model are established. The CLAUD model is identified as a representative model for our needs. A scheduling approach that tackles the characteristics specified in CLAUD is required, and in order study scheduling techniques considering communication aspects, a classification of some of the many scheduling strategies available in the literature in accordance with the communication costs is done. The classes of scheduling heuristics are: those which consider the delay as the only important communication parameter; those which define communication costs as an abstract *CPU penalty* and those in which the communication costs are considered to be a *CPU penalty* and a delay. Then, Section 3.4 highlights the fact that most heuristics are based on the ratio of computation and communication, i.e. the granularity of a *DAG* on the target machine is a concept used as an attempt to classify a given application in order to evaluate the performance of a heuristic. The optimality and applicability of a well known list scheduling technique, ETF [65], is analysed by considering communication characteristics defined in our scheduling model. As a conclusion, a new scheduling strategy is necessary for models that not only consider delay on communication, but also, significant overheads due to sending and receiving.

Chapter 4 introduces the *multi-stage approach*, given the general idea of the stream of heuristics that compounds the new strategy. Important definitions and concepts are proposed and an outline of each stage of the new approach is given.

The first stage of the *multi-stage approach* is described in Chapter 5. Task replication, in the manner defined by Papadimitriou and Yannakakis [106], is implemented in the first stage where tasks are clustered so that messages are bundled. The amount of replication is limited by a parameter that guides the clustering and a *replication* algorithm is proposed, which also contains procedures that exploit the neighbourhood among tasks in order to minimise the number of communication events. The *superdag* algorithm is then proposed, where the dependencies between clusters are specified using a set of rules. Finally, an analysis of Papadimitriou and Yannakakis' heuristic on a given instance shows that a critical number of *virtual* processors can be produced. Since our *replication* algorithm is based on their replication principle and also, due to physical limitations on the number of processors, a scheme to allocate the resulting superdag on the target network is proposed next.

With the objective to define a final schedule for the CLAUD model where the number of physical processors is specified, a *remapping* algorithm is proposed in Chapter 6, given the superdag produced by the first stage, the number of processors and the communication parameters. The *remapping* algorithm uses Brent's Principle and it consists of two phases. During the remapping process, locality is considered and still, messages are bundled if necessary, but in such a way that the communication delay is decreased. The mechanisms implemented have as their objective the minimisation of the communication costs (overheads and delay) while maximising the degree of parallelism (through Brent's Principle). An analysis of the best number of processors is outlined, considering the architectural parameters.

Chapter 7, the versatility of the *multi-stage approach* is proposed. Each rule is discussed and different orders and parameterisations are proposed depending on the target architectural model. In order to execute experiments and compare our approach with others, the *Mapping Heuristic* (MH) [41] which defines startup overhead and delay as the communication parameters is described. Also, a simulated machine is briefly described which is part of our experimental environment.

In Chapter 8, an evaluation considering the rules that constitute the *superdag* algorithm is performed. Their impact when considering the *delay* and *claud* models are characterised. An irregular *DAG* is taken as an example and experiments are carried out, considering different values for the parameters, and different orders and parameterisation of execution of the rules, so that the versatility of the proposed heuristic is testified. Th second part of this chapter consists of a series of experiments using random graphs, where the *clustering factor*, the communication parameters and the number of processors varies, so that different models and situations are taken into account. Conclusions and comparisons with other approaches are performed and evaluated.

Chapter 9 analyses regular structures such as trees and the diamond *DAG*. Firstly, the *layer* partition is defined for both the *delay* and the CLAUD models, considering as input the Full Binary Tree. Then the *subtree* partition is proposed for the Binary Tree, also considering both models. Depending on the model, one of the partitions produces an optimal schedule and it is verified that the *multi-stage approach* can produce both partitions. Experimental evaluation is carried out, considering the best results produced by the *multi-stage approach*, ETF and MH, where both predictions and time to completion of the simulations are compared. The Inverted Binary Tree is also evaluated and finally, the *diamond DAG* is considered. An analysis of the *boxes* partition for the diamond is carried out and compared with our *multi-stage approach*. A series of experiments is executed, for different values of communication parameters, number of processors, and *clustering factor* and an evaluation is presented.

The final chapter concludes this dissertation with a summary and a series of proposals for future work.

Chapter 2

Models, Machines and Misconceptions

2.1 Introduction

The need for efficient and fast solutions for the variety of applications being undertaken by scientists has enticed researchers into the field of parallel computation through the design and development of a wide range of parallel computing systems. Unfortunately, a standard model of parallel computation has eluded researchers. Given the complexity of the applications, one key software issue is the portability of these programs. Portability is achieved in the sequential computation model by virtual of the fact that von Neumann architectural model is a reasonable approximation to real sequential computers that can be used by hardware, language and system designers as well as application programmers when doing performance tuning, even though there exists a wide range of problems related to the sequential computation. Therefore, the acceptance of parallel computation is being hindered by the difficulties faced by users who wish to parallelise their applications, since they have to take into account the many distinct parallel machines.

Many attempts to define a standard model of parallel computation have been proposed [34, 56, 129]. Each model tries to incorporate the parameters of parallel architectures that are essential to characterise the different parallel machines. Since the original PRAM model was first proposed [115], many other models have been defined in an attempt to realistically characterise the actual implementations of architectures, since a more realistic model of parallel computation than the PRAM model becomes necessary.

PRAM algorithms [32, 89, 114] may have an inefficient performance when executed on real parallel machines. It can be verified through performance evaluation techniques that PRAM models are non-realistic since important characteristics of parallel computation, like communication characteristics, are not captured by these models. In order to represent more realistically parallel machines, models were proposed [4, 105, 106] incorporating critical parameters of parallel architectures, mainly the ones related to communication overhead. However, it must be pointed out the importance of PRAM models as an aid to complexity analysis of algorithms. If a PRAM algorithm is cost optimal, it may be a suitable basis for the design of a parallel program target to a real machine [20, 114].

As parallel architectures evolve (become faster) so the critical characteristics identified for a particular architecture change since performance bottlenecks are simply moved from one aspect of the design to another in the newer version. Thus, it is important to capture, in any model of parallel computation, all of these aspects as parameters in some form.

Due to the various aspects of parallel computation, different levels of abstraction dependent on different details used to describe parallel computation can be found in the literature, as for example, described by Heywood and Ranka in [59]. The *machine model* is the lowest level of abstraction and describes the hardware and operating system of the machine. The next level of abstraction, the *architectural model*, incorporates details of the interconnection network and its purpose, defining the manner in which the processors are connected: either through shared memory or distributed memory and also specifies if the computer communicates synchronously or asynchronously. Next, the *computation model* is defined to reflect the cost and resources of the parallel computer, in the most simplistic way. This level of model is used in theoretical work. Many times *computation* and *architectural models* are considered equivalents, but the *computation model* is proposed to specify the costs of the target computer, although the specification may be rather general than detailed. Finally, the *programming model* is the next higher level of abstraction and describes the computer in terms of a formalisation of a language.

The *computation model* is a bridge between the *architectural model* and *programming model* which contains properties of realistic *architectural models* and onto which *programming models* can be mapped. However, different parallel environments have been developed leading to a lack of standardisation of the *computational model*. This chapter provides an overview of existing models of parallel computation and attempts to identify the critical parameters of the different classes of parallel computers. There are models which have been proposed as general models of parallel computation, incorporating parameters that can represent the different classes of machines. Through some examples of real parallel machines, the relationship between these models and real architectures is discussed. The purpose of this review is to provide the basis for a realistic approach to the scheduling problem, by defining parameters that are essential in order to represent both the parallel architectures and also the applications that are not to be scheduled on such parallel systems.

2.2 Global memory models

We start from the first model proposed as a model of parallel computation. The simplicity of this model suggests that it is a good platform for the evaluation of the complexity of the algorithms rather than a fairly realistic model of parallel computations.

An idealised model: The *Parallel Random Access Machine* (PRAM) was first proposed in a 1978 paper written by Fortune and Wyllie [115], as a model for parallel computation. A year later, in his thesis [134], Wyllie specified the model in greater detail. The PRAM model consists of a number of processors, each associated with a local memory. Communication among the processors is carried out via shared memory with each event (a send or receive) being completed within one step. Each processor computes one instruction in one unit time, and all processors execute the same number of instructions (maybe different operations) on different data, synchronously [32, 66, 72, 114].

The great acceptance of the PRAM model by the theoretical community has been due to its simplicity and universality. A large number of parallel algorithms which execute efficiently on the PRAM model have been designed. In a continuous attempt to map out the borderline between the problems that are easy to parallelise and the problems that are hard, researchers have proposed fundamental paradigms and techniques using the PRAM model.

While the PRAM model is an idealistic model of parallel computation, unfortunately it is not a realistic one, since it does not represent many of the important characteristics of real machines. For example, in existing physical shared memory systems, the memory access time no longer remains constant when the number of processors and the size of memory is increased. PRAM algorithms are therefore, only efficient in terms of an abstract model and the majority of these algorithms cannot be executed in reasonable time on real machines since, for example, the model ignores many crucial communication issues. Nevertheless, much research effort has been expended in attempting to identify efficient techniques to simulate such algorithms on realistic machines [72, 120].

One of the difficulties which are found when trying to execute PRAM algorithms on existing shared-memory MIMD machines is that they are asynchronous. If there is a large number of processors, it would be very expensive to synchronise all of them and therefore, the MIMD machines must allow their processors to work independently. Based on the PRAM model, the asynchrony is incorporated to PRAM models as we find in the literature and briefly described as follows.

The semi-asynchronous PRAM model: Gibbons [50] was the first to introduce the asynchronous behaviour of parallel machines in his asynchronous PRAM model, considering it as a variant of the PRAM model more suited to shared memory MIMD machines. A collection of m processors, each with local memory, communicate through global memory. However, there is no global clock. The processors can read and write to the shared memory asynchronously. However, no processor may read the same location that another processor writes unless they *synchronise*. This model is said to be *semi-asynchronous*, since there exists a synchronisation step on a subset P of processors. This synchronisation step is a logical point in the computation, where each processor in P waits for the other processors in P to achieve their respective logical point. Therefore, all of the processors in P which access the same memory location, do synchronise.

Gibbons considers the synchronisation cost to be a function of the number of processors. A fixed communication delay d is considered, so that a global read takes $2d$ time and a write takes d time. Like the synchronisation cost, the delay d is a function of the number of processors m.

With the aim to eliminate synchronisation costs, other models were proposed where processors perform in a complete asynchronous mode.

The APRAM model: Considering the many available asynchronous systems and observing the need to eliminate the synchronisation overhead, Martel *et al.* [92] defined the *Asynchronous* PRAM (APRAM) model. The Gibbons *Semi-Asynchronous* PRAM model is not the same as APRAM. The APRAM is defined as a set of processors which share a global memory. Each computation, called an *event*, is executed locally or by accessing the global memory, both lasting one unit time step. Instead of assuming a global clock, the APRAM is guided by a *virtual clock* [28, 80]. In Martel's APRAM model, delays are considered to be totally arbitrary. Nishimura [101] addressed some of the asynchronous characteristics of many real machines, assuming probabilistic delays. Kruskal *et al.* [79] and Cole [28] introduced asynchronous PRAM models that assume bounded deterministic delays.

We note that the APRAM model is not a "variation" of the PRAM model, but a more general abstract model of parallel computation because it includes the PRAM model as a special case.

As a remark, Gibbons considers his asynchronous model a variant of the PRAM model, but some more realistic characteristics are considered, namely the synchronisation cost and communication delay which occurs on real machines. An advantage of the Gibbons' *Semi-Asynchronous* PRAM model over Cole's APRAM is that the latter does not account for communication delays. The incorporation of asynchronous communication in the PRAM models is very important on the attempt to represent real machines. Nevertheless, these models may be very useful for evaluating algorithms under some of the many aspects that should be accounted when considering parallel architectures. Since this is true for many models of parallel computation which are inspired on the PRAM models, we mention here the LPRAM model which address the importance of locality.

The LPRAM model: Bearing in mind that when the number of processors in a shared memory system is increased, the communication cost increases by more than a linear factor, Aggarwal *et al.* [4] proposed the *Local-memory Parallel Random Access Machine* (LPRAM) model. The LPRAM model addresses the complexity of global memory and supports the exploitation of locality to promote efficient algorithms, i.e. the effectiveness of a parallel algorithm is higher when processors compute tasks using local data, since this reduces the number of global accesses.

A LPRAM consists of a set of identical processors, each of which is associated with unlimited local memory; the processors do not necessarily execute the same code. At every time step each processor either performs an operation on at most 2 data items stored on the local memory in one computation step or writes and then reads a data item from the global memory in a single communication step. Simultaneous reads from the same location in the global memory are permitted, but only one processor at a time is allowed to write to the same cell of the global memory. In this respect, the LPRAM is similar to the

CREW PRAM [66, 72] and can be viewed as a MIMD machine whose processors cooperate in synchronised fashion.

When evaluating parallel algorithms, Aggarwal *et al.* concentrated on the number of communication events that are incurred when a given schedule S of the algorithm is executed on an LPRAM machine. Two separate quantities need to be considered:

- the computation time of the schedule, which is the duration of a computation step multiplied by the number of computation steps; and
- the communication delay, that is the number of communication steps multiplied by the duration of each communication step.

Let us denote the number of computation steps and the number of communication steps by $ncomp$ and $ncomm$, respectively. Each computation step has unit length, while the length of the communication step is equal to a certain value l. Therefore, the completion time of S is $\mathcal{M}(S) = ncomp + l * ncomm$. It is easy to see that $ncomp$ and $ncomm$ are functions of the problem size n and the number of processors m, i.e. $ncomp = f(n, m)$ and $ncomm = f'(n, m)$. Therefore,

$$\mathcal{M}(S) = f(n, m) + l * f'(n, m)$$

In [4] evaluations of various parallel algorithms such as matrix multiplication, sorting and FFT, were carried out in order to determine the value of n for which they are efficiently executed on a LPRAM machine, i.e. the value of n where $f(n, m) > l * f'(n, m)$. Aggarwal *et al.* defined l as the global memory latency [4], which may depend on m, when considering global memory systems with limited memory bandwidth. They stated that an efficient time to completion can be achieved whenever the problem size n is large enough when compared with l and m, for any given topology. Otherwise, it is suggested that topological information should be considered.

Aggarwal *et al.* stressed the importance of exploiting locality; their LPRAM model captures enough information for instances in which $n \gg m$. However, it is not explicitly specified how the data items are grouped among processors. It follows that an allocation strategy must be devised that incorporates the locality issues. The paper [4] suggests that LPRAM can reasonably model coarse grained applications, considering l and m. In any case, it must be borne in mind that the main objective of defining a model like LPRAM is to evaluate several aspects of communication complexity of the PRAMs rather than defining a realistic model of parallel computation. Overheads which are incurred on the processors when sending or receiving messages, delay due to contention are some of the communication aspects which are not captured by the LPRAM model.

Knowing that LPRAM cannot be considered a general model of parallel computation, the same authors proposed another model that captures other relevant characteristic of parallel computers, as we discuss as follows.

The BPRAM model: Usually, in global memory machines, the period of time to get the first word from global memory is quite long, but after that, subsequent words can be accessed quite fast. Considering this aspect, it is better to transfer data in blocks. Aggarwal *et al.* [3] defined the *Block Parallel Random Access Machine* (BPRAM) model, where *blocks* of data are transferred, when communication is necessary. Consequently, an efficient computation under this modes required exploitation of locality of reference.

The BPRAM parameters are: the latency (in our context, this latency is the startup overhead) l, the number of processors p and the size of the application n. Each processor is associated with its local memory, from which accesses are done in unit time. Non-local access takes $l + b$ units of time, where b is the size of the message being transferred. It is important to know that blocks that are accessed simultaneously can not overlapped, i.e. communication is done once at each time.

The BPRAM model can also represent a non-global memory system where processors communicate through message-passing if for every communication a message of size m is transferred in $l + m$ units of time between the communicating processors.

The brief description and discussion of some of the global memory models shows clearly that each one of the above mentioned models is very important in the analysis of specific aspects of parallel computation. However, these models fail when attempting to represent all of the realistic issues of parallel computations.

2.3 Distributed memory models

Clearly, the PRAM model is relevant for the exploitation of parallelism in many problems, but it is not sufficiently realistic to model real computation on existing machines. In such model, the processors communicate via global memory in a constant time (not necessarily in one cycle). Consequently, for real computations it would be difficult to increase the number of processors and the size of the global memory without hindering the performance of the system. For example, when processors are connected to the global memory via a shared bus, the system usually saturates with around 64 (or even less) processors due to the high traffic. In massively parallel processing this number cannot be considered high. One solution to this problem has been the development of the *non-uniform-memory-access* (NUMA) model, which correspond to global-memory systems in which the access time varies with the location of the memory word. Each processor is associated with its local memory, and the collection of all the local memories forms the global memory address space accessible by all of the processors [66].However, with the present generation of machines, non-local memory access takes an order of magnitude longer than the local memory access.

With the objective of designing a scalable system, distributed memory networks, the multicomputers, have become a significant area of research whose main purpose is the specification of an efficient solution for very large dimension problems. However, the performance of these proposed solutions can be affected not only by the limitation on bandwidth but also by the latency on communications and CPU overhead. Many researchers have evaluated the behaviour of distributed memory architectures, with the aim of designing a general purpose parallel model.

The Distributed Memory Model consists of a set of processors connected by links under some topology. The network can be represented by an undirected regular graph with m nodes, each node corresponding to a processor and each edge, representing a bidirectional link between two adjacent processors. Graph theory is applied in order to solve many problems related to a network of processors, like, for example, routing of messages and embedding. The main points addressed by Distributed Memory Models are:

- each processor is associated with a local memory (the memory-processor pair is called a processing element, the PE);
- a fully connected network is difficult to implement, especially when considering a large number of processors;
- the diameter of a network is the shortest path between the most distant pair of processors. When dealing with distributed networks under a specific topology, it is desirable that the diameter is kept as small as possible;
- high parallel message passing density should be supported.

Existing distributed parallel systems tend to be based on a variety of different topologies which researchers have found to have useful characteristics including good performance. There include the binary hypercube, cube-connected cycles, butterfly, shuffle-exchange, de-Bruijn, meshes and tree networks [14, 17, 32, 66, 111]. This last topology has been extensively studied due to the fact that many algorithms have a tree-like structure and an efficient execution of the application may be achieved with its minimal dilation embedding on the architecture [68, 69].

In the literature on PRAM models, see e.g. [49], it is claimed that global memory models can be emulated by a distributed memory machine [13, 83] reasonably well. For this, each local memory is considered to be a "portion" of the whole global memory and each PE is considered to be a PRAM processor. Each constant time step in the execution of the PRAM algorithm corresponds to the time to resolve the request (via routing mechanisms) on the distributed memory model. Many PRAM algorithms are classified as EREW [66, 72], that is, there exists an efficient solution of the respective problem on the EREW PRAM model. For these algorithms, if there are simultaneous requests to access the global memory, only one request is realised and efficient permutation routing techniques [49] are used to solve non-local accesses. Also, there are modern multicomputers, in which the routing of messages is the responsibility of a mechanism attached to each PE, the router, which is designed to route the messages between the two communicating PE's in the most efficient manner, decreasing the amount of work given to the processor.

In multicomputers, there need not be a tight synchronisation among the operating systems on the different processing elements when providing a common address space [2], as in global memory architectures. By contrast, in a multiprocessor in which a single bus connects processors with a global memory, the traffic on the bus will tend to be much higher than it would be when using an interconnection network, because it is used for cache coherence rather than just explicit interprocessor communications. In message-passing systems, computation may be overlapped with communication, with the communication of fewer but large blocks of messages to overcome the high startup overheads. Note that there have been the development of NUMA architectures where a distributed shared memory is implemented, incorporating the scalability of message-passing systems and higher programmability. The lack of programmability of multicomputer systems is attributed to the fact that programmers are imposed to distributed efficiently their applications on the machine [66, 110].

In attempting to address the issues related to the communication cost in distributed memory systems, a couple of models merit discussion. Actually, these two models cannot be considered representative models of parallel architectures, but they incorporate and evaluate some of the many important characteristics in parallel computation.

The delay model: Papadimitriou and Yannakakis defined the *delay* model for parallel computers, in which the *delay* on the communication between two processors is taken as a crucial parameter of parallel architectures [106]. The model considers an unbounded number of identical processors capable of executing *tasks* and whenever two processors communicate, there is a delay from the time the data item leaves the source processor to the time that the data is available in the destination processors. The communication delay is denoted by τ and the processor network is assumed to be a fully connected one, in the sense that data item becomes available on any processors after τ time units. It should be noted that although the *delay* model is classified here as a distributed memory model, it can also model systems with global memory where accesses are resolved in τ time units.

Although this model includes one of the important characteristics of parallel architectures, there are still certain classes of parallel machines which have other characteristics that cannot afford to be neglected in a realistic model. One such characteristic, the startup cost, will be discussed throughout this chapter.

Another model is also considered relevant where delay is also specified, but in a different way. Also, an abstract representation of the startup overhead which is incurred in the communication processors is introduced.

The event model: Papadimitriou and Ullman evaluated the tradeoff between communication and computation when scheduling an application modelled by a *DAG* $G = (V, E)$ on a set of processors [105]. They defined the *event* model, in which each processor is associated with a very large local memory and each task of the application is scheduled on one (or more processors) at a given time, given that all of the predecessors of a task have been computed previously.

Papadimitriou and Ullman define the computation cost of a schedule as the measure of the computing time of the schedule when communication delays are ignored. The communication is modelled taking into account the number of communication *events*. Let $(v, u) \in E$ be an edge in which the vertex v is allocated to a processor p and vertex u is allocated to a distinct processor

p'. A dependence relationship D between processors allocated to adjacent vertices is established, so that there exists a dependency edge from processor p to processor p' where the vertex u can only be executed on processor p' after p finishes the execution of v. The number of edges in D represents the communication cost of the schedule of the *DAG* on the set of processors. Therefore, the set D corresponds to the number of sending events incurred due to communication between different processors. The relation D can be visualised as a *DAG* in which the vertices correspond to the processor-task pairs and the edges represent the communication events which occur among processors. The *communication delay* d in the *event* model is the longest path in the *DAG* D. The value d represents the amount of time the processors are delayed due to communication incurred in the schedule of the *DAG* G. Therefore, events that occur concurrently are counted as one.

On the evaluation of the tradeoff between communication and computation when scheduling a *DAG*, Papadimitriou and Ullman addressed the high startup cost incurred when the set of processors is linked by a local area network. Also, in such a network, the latency due to communication between distinct processors is small. Regarding a network with such characteristics, the number of communication events must be minimised.

The evaluation of particular schedule of a *diamond DAG* on a network of processors based on the *event* model leads to the conclusion that achieving an efficient time to completion does not necessarily correspond to the minimisation of both communication and computation costs, but to find their tradeoff. These results are important when executing a great number of problems on parallel machines.

Similarities between LPRAM and the *event* model can be found. Papadimitriou and Ullman's approach gives emphasis to distributed memory systems while the LPRAM model addresses global memory multiprocessors. Aggarwal *et al.* observe that Papadimitriou and Ullman's model can be seen as a pipelined version of the LPRAM model.

We can not ignore the importance of the *delay* and *event* models characterised [105] and [106], respectively. However, there are many realistic issues which were not capture by both proposals. Their importance may be highly attributed in the complexity analysis of algorithms, due to their simplicity.

In [106], Papadimitriou and Yannakakis proposed a scheduling heuristic. The method can be considered as a technique to be incorporated in more advanced scheduling strategies rather than a pure realistic scheduling approach, since there are some realistic issues that should be considered, such as number of physical processors and overheads due to sending and receiving of messages.

In [105], the models were proposed with the objective to address issues that should be considered when devising parallel techniques. The delay defined by Papadimitriou and Ullman is a global concept of delay when the application is parallelised and it would be difficult to incorporate such definition in a scheduling heuristic where usually, local decisions are carried out. Also, the startup overhead defined represents message traffic in order to characterise contention rather than to characterise *communication events* which are incurred during the parallel execution of the application. In respect to this latter communication characteristic, only the events affecting the critical path processors need to be used economically.

Other models were proposed in the literature with the objective to study specific aspects of parallel computations. Models such as XPRAM of Cosnard and Ferreira [31] which was proposed as an extension of the PRAM model, in order to analyse topological aspects on distributed memory machines. Cosnard and Ferreira stated that the PRAM model can be seen as a fully connected network. This view is extended to XPRAM, where X is any arbitrary interconnection network. The overhead due to data routing is defined in XPRAM when a PRAM algorithm is executed in such model.

In the attempt to represent a wide range of parallel machines, general models of parallel computation are proposed. We shall discuss some relevant models at next.

2.4 General purpose models

The existing models which propose to represent parallel architectures, generally aim to incorporate the main features of the three classes of parallel machines: global memory multiprocessors, distributed memory multicomputers and networks of workstations. These models attempt to capture the essential parameters of parallel computers, making parallel machines and their systems easier to use.

The absence of a standard model of parallel computation and the lack of uniformity among parallel computers influenced many researchers who attempt to establish a bridge between parallel applications and parallel machines in an architecture-independent style, that captures the essential and significant features of the machines while abstracting away details that differs among distinct types of parallel machines.

The BSP model: Valiant [129] defined the Bulk-Synchronous Parallel (BSP) model of computation in an effort to capture the most relevant characteristics of parallel architectures, while abstracting away from the concept of network topology, which is not standard between different architectures.

A BSP computer consists of a set of PEs, a communication network and an efficient barrier synchronisation mechanism for all processors (or a subset of processors). The parameters which characterise the model are:

m: the number of processing elements;

s: the processor speed, defined as the number of floating-point operations that can be executed during one second;

l: the minimal interval between two synchronisation events;

g: the ratio of the total number of local operations performed by all processors per unit of time and the total number of words delivered by the communication network per unit of time.

The main feature of BSP is the concept of the *superstep* which is defined as a sequence of steps followed by a synchronisation. The execution of an application under the BSP model is a succession of *supersteps*, each of which consists of computation of tasks and message delivery between processors. During a *superstep*, all of the data necessary for execution of tasks by a processor must be locally available: any required data must have been fetched already from non-local memories. After all processors have executed their respective *superstep*, the *barrier synchronisation* is called and all processors are synchronised, enabling the initiation of another *superstep*. Figure 2.1 shows a *superstep* between two *barrier synchronisation* calls. During the *superstep*, computation is performed and communication is carried out until the second call for the barrier.

The parameter g represents the global cost on the communication, incorporating congestion and latency. Suppose that every processor has the same communication pattern and can send and receive up to h units of data, defining an h-relation among all processors. Therefore, in the BSP model the upper bound on the global communication is gh time units.

The *minimum* size of a *superstep* is the minimum time between successive global synchronisations, represented by l and it is determined by the maximum frequency at which the "barrier synchronisation" can be performed by the system [93]. The *maximum* size of the *superstep* is a property of the application and it depends on the amount of work done between successive synchronisation points, being also determined by the requirements for parallel slackness. Let the maximum size of a *superstep* in time units be denoted by $\mathcal{S}$ and w be the maximum number of local computations performed by any processor. The upper limit for $\mathcal{S}$ is $w + l + gh$. This upper limit can be smaller depending on two situations described below:

- when synchronisation is totally independent on the communication system:

$$\mathcal{S} \leq \max\{l, w + gh\} \tag{2.1}$$

  An example of this case is the CRAY T3D (Subsection 2.5) which implements the barrier synchronisation in hardware and it is independent of the network topology.

- when synchronisation is performed by means of the communication system:

$$\mathcal{S} \leq w + \max\{l, gh\} \tag{2.2}$$

The order in which the local computation of tasks and the delivering of messages is performed by a processor is not stated. Thus, while a message is being sent, the processor can execute tasks allocated to it during the current *superstep* if the necessary data is locally available, exploiting any *parallel slackness*. The BSP model captures the main communication characteristics of a network through the parameters l and g. The model works better on machines for which the values of these parameters are low so that the time to completion of the computation performed in a BSP machine is minimal. Setting $l = g = 1$, the BSP model is reduced to the PRAM model.

Analysing the parameter g in more detail, consider m processors able to execute s floating-point operations per second. Suppose that b words can be delivered *to all processors* in one second, i.e. the *global bandwidth* is equal to b words per second. Therefore, each processor bandwidth is up to b/m words per second. Taking into account the h-relation, each processor takes

$$\frac{h}{b/m} = \frac{hm}{b}$$

seconds to communication h words, an amount of time equivalent to shm/b floating-point operations. As the g parameter is the ratio of the global computation throughput (ms) to the global communication throughput (b), its value

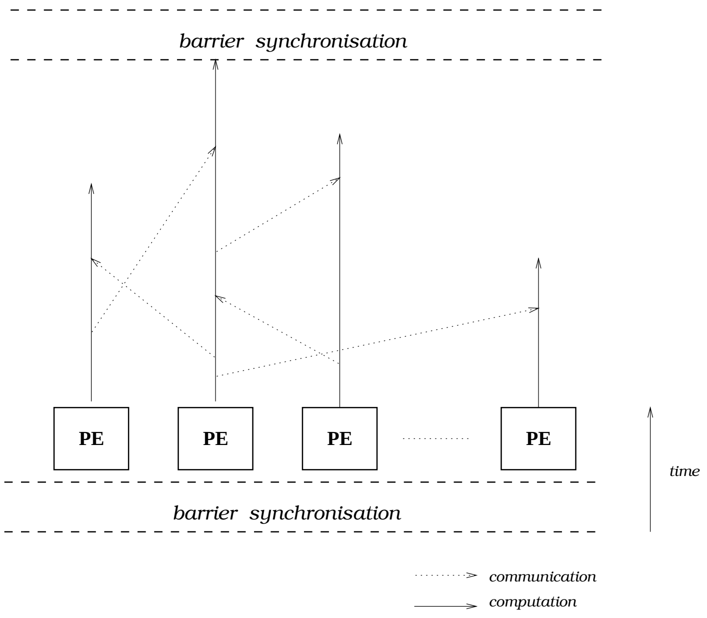


Figure 2.1: *Superstep*

is at most $\frac{ms}{b}$. It must be noted that the value of h is greater than a certain lower bound, otherwise if small messages are communicated, the execution of the application on the given system will be hindered by the startup cost, which makes communication involving many short messages slower than communication involving a few longer ones.

The g parameter may depend on characteristics of the system like the number of processors, the processor speed and the number of words that can be delivered per second.

The BSP model aims to provide a unified framework for scalable computing and a portable representation across all parallel architectures (distributed memory multicomputers, global memory multiprocessors, networks of workstations). Not only has a high-level parallel programming model using message passing been proposed, but the design of parallel programs and global memory systems has been suggested [95].

However, the BSP model fails in the abstraction of some machines like networks of workstations, because it does not include information about the startup overhead, a cost that is critical in such machines. The model assumes that the messages exchanged between processors are large enough so that startup overhead can be neglected. But even if only large messages are considered, ignoring details of the interconnection topology may affect the execution of the given application on the multicomputer.

It is not always the case that a parallel machine behaves in the way specified in the BSP model. The non-blocking communication during a *superstep* may not occur in a machine such as network of workstations of a network of transputers. Also, the synchronisation mechanism can be expensive to implement. When synchronisation depends on the communication system as expressed in Equation 2.2, the network has to provide a strategy to synchronise all processors, which could lead to a high synchronisation cost.

Many characteristics of the architecture are all together represented by the single parameter g, like communication bandwidth, latency (as usual, the latency is given by the diameter of the given topology, which is an average measure) and contention. The BSP model may require powerful and expensive communication hardware to support the PRAM model [57]. Also, latency is still an important parameter such that one may not assume that small values are assigned to it. Due to physical limitations, to hide latency may become practically impossible, as discussed in [17].

The LogP model: Although efficient processing elements and interconnection networks have been developed, the interface between both of these components has long been the stumbling block to taking advantage of their capabilities. Increasing the number of "efficient" processing elements does not necessarily decrease the execution time of a given application, due to communication costs incurred. In an attempt to specify a realistic computation model, Culler *et al.* [34] introduced the LogP parallel computational model.

As said by Culler *et al.* in [34], "the BSP model was one of the inspirations" for the specification of the LogP model. However, the BSP model does not specify the startup overhead, which, as stated in [34], is a significant value in many parallel computers, even when compared with latency (see Section 2.5 for some measures). The importance of this parameter is based on experiments on real machines, particularly, network of workstations, a class of parallel environment that cannot be ignored.

Furthermore, the main characteristic that differentiates the LogP model from BSP, is the asynchronous behaviour of the former. There is no *barrier synchronisation* in the LogP model and no specification of a *superstep*. When processors complete their job (computation of tasks and message delivery), they can be allocated to tasks from another thread, even if there exist processors in which communication and computation are still to be executed, in contrast to BSP, where the processors remain idle, waiting for the barrier mechanism to be resumed. The asynchronous aspect of the LogP model can be considered an advantageous characteristic, if one takes into account the fact that most parallel computers perform in an asynchronous fashion [96].

As in the BSP model, LogP is proposed to be a computational model in which global characteristics of parallel architectures are represented, but again details concerning the topology of the network are not considered. The LogP parameters that characterise the communication between processors are as follows:

m: the number of processing elements,

l: represents the latency when distinct processing elements are communicating,

g: the minimum possible time interval between two consecutive sending or receiving events performed by a processor,

o: the startup overhead associated with the sending/receiving of messages to/from a processing element.

The maximum number of messages in transit between any two processors at any time is $\lceil l/g \rceil$. The maximum latency of a given topology is the asymptotic average distance between processors; we note that it is difficult to know details about the network if only a representative value is associated with it.

In order to illustrate the parameters defined in LogP, the P-way *broadcasting* of information to m processors is reproduced from their example discussed in [71]. A data item is broadcasted from a source processor p_0 to the other $m - 1$ processors, each processor when receiving the data, immediately sends it to its neighbours processors. When starting to send the data to the first of its neighbour p_1 as seen in Figure 2.2, the processor p_0 incurs the startup overhead of o time units. After that, the data reaches p_1 after l time units, where o time units of overhead is incurred. At time step $l + 2o$, the processor p_1 can start broadcasting the data to its adjacent processors yet to receive the data. Meanwhile, processor p_0 can send the data to its next neighbour, processor p_2 g time units after the data is launched in the link between p_0 and p_1. If $g \leq o$, the gap is completely overlapped by the startup overhead o. Given Figure 2.2(b) shows the Gantt Chart for the broadcasting of a data item through a given network of processors in Figure 2.2(a). Note that only events related to communication are being considered during the broadcasting.

When evaluating the communication time for a message of, say, L-bit length, across a network, an expression for values of l, g and o can be specified. Suppose that the channel width of a network is W, the average number of hops between

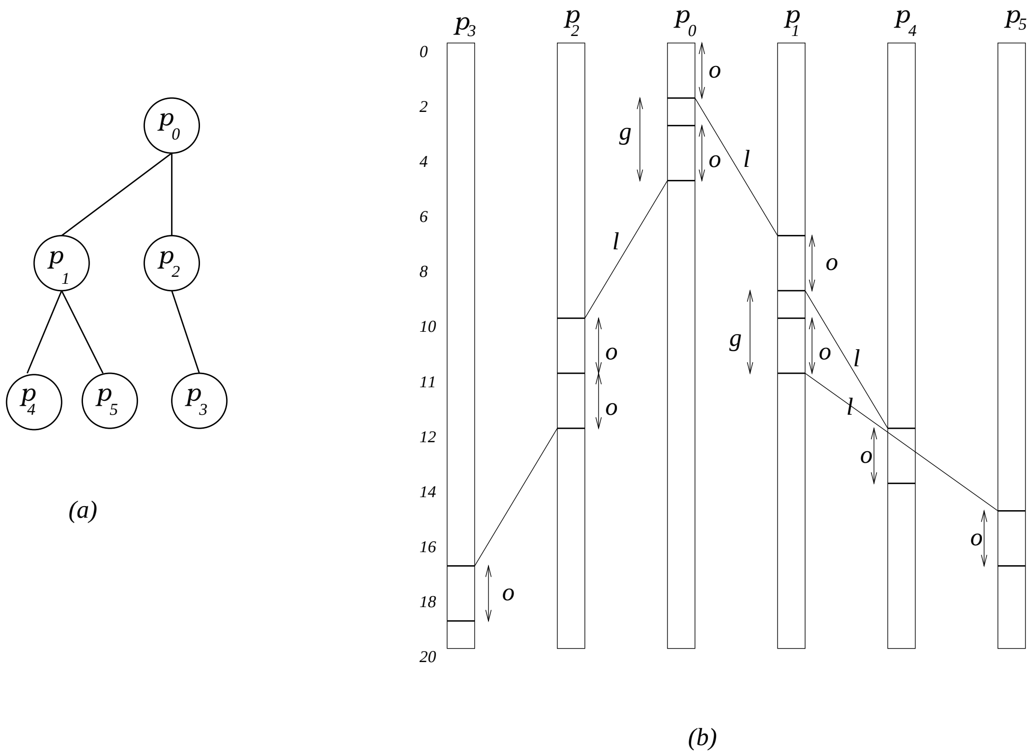


Figure 2.2: Broadcasting tree considering the parameters of the LogP model to be: $m = 6$, $l = 5$, $o = 2$ and $g = 3$

source and destination processors transmitting the message is h and the delay between two adjacent processors is τ. Let t_s be the startup time of a processor when sending a message. The last bit of this message will take $\lceil L/W \rceil$ cycles to be introduced to the channel. Then, this same last bit will take $h\tau$ time to cross all intermediate processors and channels until the destination processor is reached. Finally, the destination processors will spend t_r time preparing to receive this message. Thus, the total communication time of the L-bits message is $t_s + t_r + \lceil L/W \rceil + h\tau$ and thus the values for the LogP parameters can be defined as:

$$o = (t_s + t_r)/2$$

$$l = \lceil L/W \rceil + h\tau$$

Considering the bisection bandwidth per processor xW/m, in which x is an integer that varies between the different topologies, the parameter g is defined in [34] as:

$$g = \lceil Lm/xW \rceil$$

The assumed values of all these parameters were based on average measurements taken from a given parallel computer [34], so that a realistic approach could be specified, resulting in an abstraction from details of characteristics of the network.

In order to validate the importance of the parameters of the architecture addressed in the LogP model, Patterson [109] presented some valuable values taken from a variety of machines, listed in the Table 2.1. Patterson in his talk

Machine	l	o	g	m
HP-720(Ethernet)(PVM)	50	2400	$80 * m$	4 to 16
HP-720(FDDI)	4	400	$2 * m$	8 to 32
DEC 5000/ATM	20	20	20	8 to 32
Intel Paragon	1	250	2	up to 500

Table 2.1: Values for the LogP parameters, in μseconds, when given some real machines

showed, through the results in Table 2.1, the importance of the architectural parameters specified in LogP. We can see that in all the machines illustrated in the Table, the significance of the startup overhead, a parameter that is usually completely neglected in most of models of parallel computation. In systems within an Ethernet, this parameter degrades the performance of the systems, if measures to overcome its high value are not taken.

The asynchronous behaviour of many real parallel machines is well represented by the LogP model, and also, the relevance of critical parameters such as the startup overhead is addressed, providing a powerful representation of networks of workstations (an emerging popular class of parallel machine and trend for the future of massively parallel computation).

However, as addressed in the BSP model, the abstraction of the topology and the adoption of average measures for parameters may cause certain applications to be mapped to the parallel computer in an unreasonable fashion. In the LogP model, the latency parameter corresponds to the diameter of the interconnection network, which may be high, especially when considering massively parallel processors. Scheduling techniques, like list scheduling, consider in their cost function both communication costs (like *latency*) and computation costs (such as execution costs associated with tasks). When communication costs are higher than computation costs, the scheduling algorithm tends to allocate tasks to a smaller number of processors. In fine-grained applications, the inherited parallelism and locality would not be exploited well, leading to a schedule that may be far from the optimal one.

Extensions to BSP and LogP: Extensions to the LogP model can be found such as the LogGP model [6], which analyses the cases in which short and long messages are transmitted between processors of a parallel machine. A new parameter G is added to the LogP parameters, which is defined as the time per byte for long messages. The available per processor communication bandwidth for long messages is than $\frac{1}{G}$. The sending of a message of w bytes takes $2o + (w-1)G + l$ units of time to be completed, knowing that the sender and receiver processors are busy with the communication during o unit times each. The LogGP model is proposed as a better representation of "bulk transfers", since many existing parallel machines support the transmission of long messages efficiently, achieving higher bandwidth (e.g. SP2 [12], Paragon [112], Meiko CS-2 [15], nCUBE/2 [118]). Note that the ratio $\frac{g}{G}$ indicates the gain that is obtained by grouping small messages into larger ones. Therefore, locality must be exploited so that a reasonable gain is achieved.

The C^3 model was proposed by Hambrusch and Khokhar [56] in an attempt to evaluate the complexity of computation, the communication features and the congestion due to communication, for a given parallel algorithm and target architecture. The set of parameters of the architecture incorporates parameters from both BSP and LogP models, together with characteristics of the application and parameters needed to characterise congestion, as described as follows:

m: the number of processors;

l: the latency of the communication network, defined as the average distance between two processors multiplied by the latency of the communication incurred on a link;

b: the bisection width of the communication network (represents both processor bandwidth and network bandwidth);

o: the startup overhead due to the sending of a message. There is no overhead specified when a message is received by a processor;

s: the size (number of bytes) of a unit packet.

As in the BSP and LogP models, the average distance values for latency are considered to be sufficient to represent any topology. Each processing element consists of a processor and an unlimited local memory.

As in the BSP model, the C^3 model assumes that communication is synchronised by a barrier, partitioning the algorithm into a sequence of *supersteps*, each of which consists of local computation and sending and receiving of messages. The amount of computation performed within a *superstep* is normalised by the latency parameter l. The C^3 model is of relevance to "coarse grained" applications, which are those applications with average computation greater than the average communication, so that, the performance of computation is not hindered by a large communication cost.

A complete description of C^3 is beyond the scope of this thesis, but the main novelty of C^3 is that it captures the complexity of the potential *congestion* arising during communication. *Congestion* is a global event, difficult to predict because it is independent of the routing mechanism, but dependent on the amount of data communicated, the number of processors involved and the architecture topology. Suppose np is the number of processor pairs communicating and L is the number of messages routed between processors. The *congestion over links* is given by $L * \lceil \frac{np}{b} \rceil$ and the *congestion at the processors* is $L * \lceil \frac{np}{m} \rceil * l$. The term $\lceil \frac{np}{m} \rceil$ represents the number of messages at a processor when the communication begins. When crossing the route towards the destination processor, L messages compete for resources at each of the $l - 1$ intermediate processors, which are slowed down by a factor of $\lceil \frac{np}{m} \rceil$.

The C^3 model evaluation [56] shows how important and useful is to an application characteristics such as the length of messages communicated between processors. If small messages are sent ($L < s$), the communication cost is dominated by the architectural parameters and therefore, parallelising such application may not be the best solution. Taking into account the fact that the C^3 model's objective is to represent coarse grained applications on multiprocessors and multicomputers, it will be necessary to transform fine-grained applications into coarse-grained ones. If this model is to be adopted as the basis for scheduling techniques, more information regarding the application and further details about the architecture will be necessary, so that general applications can be mapped to parallel computers efficiently through the exploitation of parallelism and locality. The C^3 model ignores overhead due to the receiving of messages. If this model is used as a scheduling model, it could produce schedules that are sub-optimal because a critical path processor may receive a large number of messages and experiment results in [63] shows that receive overhead can be a significant communication parameter.

The general models of parallel computation discussed up to now incorporate relevant issues related to the communication characteristics of parallel machines. An emergent class of parallel computers in which each "processing element" consists of a small network of processors may not be fairly represented by the above discussed models.

The HPRAM model: In the attempt to define a simplistic model that represents real systems, Heywood and Ranka [58, 59] proposed the *hierarchical model of parallel computation* (HPRAM). It consists of a dynamically configurable hierarchy of synchronous PRAMs. The collection of individual synchronous PRAMs operate asynchronously from each other.

The view of the model is a hierarchy. In a P-processor HPRAM it is necessary to define "levels" of computation. At the first level, a single synchronous P-processor model is viewed. All the HPRAM instructions are the same as in the PRAM model, but one instruction is added: the partition instruction, which partitions the P processors into disjoint subsets and assigns a PRAM algorithm to each subset. The hierarchy relation defines how the PRAM algorithms synchronise with each other; this relation can be represented as a tree. At each level i, partition instructions can be used to create the next level of the hierarchy. This recursive process may continue until a PRAM algorithm is running on a 1-processor PRAM. Each sub-model of the HPRAM can be one of the variants of PRAM, except the asynchronous PRAM model. Sub-PRAMs at the same level have no access to each others block of shared memory.

The main costs associated with the communication are the latency and synchronisation costs, each of which is a function of the number of processors. Two variants of HPRAM can be defined. In one case, the partition instruction divides the shared memory among the processors so that each sub-PRAM has its own "private" continuous block of shared memory. In other case, a "shared" HPRAM is defined, where the partition instruction does not divide the memory among the processors, such that the sub-PRAMs share a space of the whole global shared memory.

If two processors are at the same level and belong to two distinct sub-PRAMs (and therefore, execute asynchronously) and they need to communicate (i.e. need to access a common memory location), they synchronise in order to avoid inconsistencies in the program that may be caused by asynchronous communication. The sub-PRAMs synchronise with each other after the end of the execution of their algorithms. To achieve this, the control returns to a previous level in the hierarchy where a partition instruction is followed by a synchronisation step.

HPRAM assumes that latency and synchronisation costs are functions of the number of processors amongst which communications and synchronisations occur in the sub-PRAMs and are defined by the architectural model. Consequently, the communication and synchronisation costs can be controlled and decreased by the partitioning into a smaller subset of processors. The problem is to know how to properly partition the processors, so that locality is exploited and communication is isolated into independent neighbourhoods. Let $l(P)$ and $s(P)$ denote the communication and synchronisation costs, respectively, considering P processors. A computation step takes unit time, as in a PRAM. The cost associated with a `partition` step depends on the number of sub-PRAMS that it creates. Let the cost of the algorithm running on the qth sub-PRAM at level, say, $l + 1$, be denoted by $C_{HPRAM}(q, l+1)$ and let Q be the number of sub-PRAMS at level $l+1$. Then, the cost of a partition step at level l that generates the Q sub-PRAMs at level $l+1$ is $\max_{1 \leq q \leq Q} C_{HPRAM}(q, l+1)$. This brief description shows that the evaluation of the performance of a HPRAM computation is done recursively.

The HPRAM model incorporates ideas properties of the PRAM model but more realistic issues are characterised, since communication and synchronisation costs are provided. The *shared* and *private* variants definition [58, 59] are a more realistic representation of existing programming models, classifying the HPRAM model as a higher level of abstraction than the BSP model. Although the authors claim that HPRAM to reflect a variety of existing and proposed architectural models, HPRAM seems to be more suitable to model global memory architectures. HPRAM enforces the exploitation of locality in applications considering delay and synchronisation costs but does not account for other important characteristics of parallel architectures, such as startup overhead.

2.5 The machines and their models

So far, we have described various models of parallel computation that have been proposed over the last 18 years. We have hinted at the strengths and weakness of these models by reference to the performance characteristics of real parallel computers. We now focus on the description of a representative sample of such machines. This will allow us to identify the architectural parameters and critical characteristics of computations which could affect the performance of the parallel machines. Through the description of real machines, we attempt to outline a correspondence between the different models and different machines, thereby exemplifying the importance of the addressed characteristics.

There is some resemblance between PRAM computations and the Single Instruction stream over Multiple Data stream (SIMD) machines with synchronised shared memory [66, 111], although the PRAM model does allow each processor to execute different set of instructions. However, at least for the present, MIMD hardware has won the battle against SIMD hardware [107]. Consequently, the machines described are all MIMD.

Albeit symmetric multiprocessing (SMP) systems based on global memory are in widespread use, the parallel computing community places much emphasis on the design of scalable highly-parallel processing systems which can efficiently execute any class of problems. Although for many years, processor technology has continued to improve dramatically especially in terms of performance, it is generally agreed that it is more effective to build a distributed memory MIMD machine with many single-chip uniprocessors than a high-performance SMP machine [111]. It is also common belief that message-passing systems are the ones most likely to achieve a higher performance. However, proponents of the shared memory systems claim that such systems form a promising class of parallel architectures with important applications for which its programming model is preferable. Examples of *Shared Memory* MIMD *Systems* are the BBN Butterfly, the Ultracomputer and the Convex SPP1200 System. Distributed multicomputer that are currently available include the commercial-application oriented AT&T 5100 and Tandem Computers Inc Himalaya; and the scientific- and technical-application oriented CRAY T3D and the Intel Paragon.

Among a wide range of parallel computers, some machines have been chosen here in an attempt to exemplify global memory multiprocessors and distributed memory multicomputers. Although the Ultracomputer is not a recent machine, it is however one of the last *real* shared memory multiprocessors still in use. The Convex SPP1200 System is claimed to be a shared memory machine, but actually it emulates shared memory very efficiently on a physically distributed memory. The SP2 and CRAY T3D systems are declared to be multicomputers. The designers of the CRAY T3D found the only way to implement barrier synchronisation efficiently was to do this in hardware. The J-machine is an academic multicomputer, being used as a platform for software experiments in fine-grained parallel programming.

The Ultracomputer: The Ultracomputer machine, developed by Gottlieb [51, 53], was proposed to be a highly general purpose parallel computer, scalable and able to support both high-level languages and a multiprogramming environment. This global memory MIMD machine is mentioned here because its designers are amongst the few researchers who claimed, while developing it, that the future of parallel computing lies on global shared memory systems. Also, it is an example of a machine where the processors are at one side of the interconnection network and the memory modules, on the other side.

Gottlieb *et al.* [51, 52] claimed that , it is very expensive to construct a fully connected network to provide uniform (fast) access to non-local memory in message-passing systems. Therefore, considering different topologies of the interconnection network, a routing mechanism must be implemented in a distributed memory system. On the other hand, Gottlieb defended that in a computer with global memory, all PE's view the memory as a single module and "faster" access may be carried out. Although such systems are known to be non-scalable, Gottlieb attributed the scalability of this global-memory MIMD machine to the implementation of the *fetch-and-add* [51, 66, 79] mechanism in the switches of the Omega network, which links 2^k (for some k) processors to the global memory (composed of modules).

The relevance of the Ultracomputer machine is that it is one of the latest examples of a truly massively parallel, yet shared memory machine. The design of the Ultracomputer was based on the WRAM model [51, 52, 53] which incorporates a powerful mechanism such as *fetch-and-add* [66]. However, since the mechanisms are solved at the switch points, its potential can be better achieved if faster and cheaper switches are developed.

When considering the Ultracomputer with m processors, the interprocessor communication time is the same for any pair of processors is the same for a unit data item. In this case, the so-called general purpose models of parallel computation like BSP, LogP, C^3 and even HPRAM would provide a suitable bridging model for such a machine due to the fact that the values assumed for their parameters are average measures.

Convex Systems: The Scalable Parallel Processing (SPP) family is the first generation of multi-purpose parallel computers developed by Convex Systems. The Convex SPP1200 System, a MIMD global memory system, contains a set of Hewlett-Packard PA-RISC processors, currently varying from 2 to 128 processors. A hierarchical memory subsystem facilitates the access from any processor to any memory location of the system [21]. The global shared memory (GSM) is proposed to provide the scalability benefits of distributed memory and automatically maintains complete data coherency through the system. The degree of data movement is decreased due to dynamic distribution of data done by the GSM.

The SPP1200 supports a two-level hierarchy of memory. The first level consists of a traditional memory crossbar interconnect that supports up to eight processors sharing a common physical memory. This set of processors and memory model, called a *hypernode*, is a symmetric multiprocessor (SMP) with its own CPUs, memory and I/O subsystem. Each *hypernode* has a low latency shared memory, supporting fine-grained parallel applications, as processors within a *hypernode* are tightly coupled [11].

The *hypernodes* are connected using the ring topology, establishing therefore the second level of the memory subsystem. The set of *hypernodes* support coarser-grained parallelism and communication can either be carried out via shared memory or explicit message passing mechanisms.

The Convex SPP1200 system supports global memory in hardware. Cache locations may be automatically copied between *hypernodes*, without software intervention, and therefore, low overhead when using shared memory programs may follow. A low-latency interconnect, called CTI, connects many *hypernodes* providing access to global shared memory. Explicit message-passing applications may also use this interconnect for inter-*hypernode* communication [10].

The current version of the Convex SPP1200 System with 128 processors seems to be very promising when considering the fast communication system. However, this version cannot be regarded as a MPP system considering how relatively few processors it has. It is also necessary to evaluate the performance of such a systems when a significantly number of *hypernodes* are considered.

The HPRAM model presupposes a hierarchical system like the SPP1200. In a *hypernode* (the lower level in the hierarchy), parameters like latency, startup overhead, communication bandwidth and number of processors have different values when compared with the ones for the interconnection of the *hypernodes* (the higher level in the hierarchy). Therefore, the other models already discussed in this Chapter do not seem to address hierarchy, so that the *hypernodes* are likely to be considered by these models as just processing elements.

Although no independent performance studies have been published (to the authors' knowledge), the conclusion from industry is that, at present, non-local memory access time is at least an order of magnitude slower than local memory.

SP2 systems: The IBM SP2 was designed to be a general-purpose scalable distributed memory supercomputer, aiming to be capable of executing efficiently both scientific and commercial applications. The system uses an off the shelf microprocessor and a conventional operating system, so that high portability of software and applications can be achieved.

There is no guarantee that the available network technologies (like for example, FCS and ATM) perform well when considering scalable parallel systems with hundred of processors, except where special software supports communications. However, an important issue is the the cost-effectiveness of the desired system, so instead of taking an off-the-shelf interconnection network, a powerful network technology was proposed where remote access is carried out via message-passing. The current available SP2 systems contain from 2 to 512 processing elements, each of which consists of a memory and a POWER2 technology RISC System/6000 processor [2].

The interconnection topology by which the SP2 nodes communicate is called the "High-Performance Switch". The topology is the Omega network, allowing the bisection bandwidth to scale linearly with the size of the system. The choice of a multistage network was due to the logarithmic increase in the number of links per PE and the number of hops when the number of processors is increased. In direct networks there must be a software mechanism which attempts to discover opportunities to exploit locality in the parallel application and which minimises the number of hops required for each non-local access. Locality is still an issue when using the High-Performance Switch under multistage topology, but the minimisation requirements are easier to achieve [2]. A total of 8 switches are grouped in a *switch board*, which is connected to the PEs through an *intelligent Micro Channel Adapter*, which offloads work from the processors when communication is carried out. This may imply that the overheads for sending and receiving are low values.

In Table 2.2, the performance of the communication subsystem is reproduced from [2]. The latency corresponds to the time to send small messages between two processes allocated to distinct PEs, taking into account all hardware and software included between the communicating processes (therefore, processor overheads are accounted for). Table 2.2 shows different node denominations. In the SP2 system the difference of the nodes depends on their performance, configurations, memory and physical sizes. UDP/IP is a communication protocol normally used by LANs.

The switches are placed 4×4 in a switch board, providing at least one more stage for fully connectivity. This extra stage defines 4 paths between every pair of processing elements, minimising congestion in the switch. The SP2 is designed to be a message-passing system and to explore the asynchronous characteristics inherent in the application, so that computation is overlapped with communication. Therefore, of the general purpose models of parallel computa-

Node	Latency (μs)	Point-to-Point Bandwidth(MB/s)	Exchange Bandwidth(MB/s)
Thin, user space	40	35	41
Thin node2, user space	39	36	48
Wide, user space	30	36	48
Thin, UDP/IP	312	36	48
Thin node 2, UDP/IP	270	12	16
Wide, UDP/IP	269	12	16

Table 2.2: SP2 communication subsystem performance.

tion discussed in this chapter, the LogP model seems to be the one best able to capture the behaviour of this machine.

The CRAY T3D system: In its early days, CRAY Research was preoccupied by the development of parallel vector processors, like the CRAY C90 systems. Later, the massively parallel processors (MPPs) concept caught on, but one of the major challenges was (and still is) to enable a wide range of applications to sustain performance levels higher in these machines than on parallel vector processors. The CRAY T3D is the first MPP system developed by CRAY research with the objective of achieving high performance, portability and usability, although one should observes that scientific computation seems to dominate the range of applications being executed on it.

A typical MPP machine is a set of fast processors, each with local memory defining a processing elements (PEs). The PEs are interconnected under some topology. In reality, the access to the local memory is faster then accessing a remote location.

The CRAY T3D is a MIMD multicomputer with distributed physically memory and *logically shared memory*, i.e. a shared memory is emulated on a distributed environment, by the use of a router installed together with each processor to handle interprocessor communication through the manipulation of a wormhole routing mechanism, thus offloading this responsibility from the respective processors. Each node of the interconnection network contains two independent PEs, each of whose core processor is a DECchip 21064 Alpha AXP microprocessor. The topology of the interconnection network is a three-dimensional (3D) torus. The designers of the T3D chose these characteristics so that higher scalability, higher bandwidth and low-latency could be provided to a wide range of applications [76].

Although many improvements have been achieved in the communication system, the exploitation of locality for a given application is an issue that requires attention. A good balance between the amount of work executed by each processor and the communication overhead can be achieved due to the communication system and a fast synchronisation mechanism, implemented in hardware, which is similar to the *barrier synchronisation* as defined in the BSP model.

The implementation of the barrier synchronisation in hardware highlights the high cost of implementing such a mechanism in software. The BSP model is able to provide a reasonable description of this machine due to the specification of such a synchronisation mechanism. Although the compiler incorporates strategies to minimise the overheads due to remote access, attempting to prefetch data from remote PEs, it is still the case that locality of reference has to be exploited by the user.

Fred Howell [63] performed a series of experiments using the CRAY T3D MPI and results where produced to help to define a platform for his performance predictions. Some measurements provided by Howell's experiments are reproduced in Figure 2.3. In the first experiment reproduced in Figure 2.3(a), the primitives $send$ and $receive$ are executed on distinct processors, just after a barrier synchronisation is performed so that both primitives start executing at the same point. For small messages (some integers) both processors were busy executing the communication primitives for $70\mu s$. On the other experiment reproduced in Figure 2.3(b), the receiver processor was idle for some time much greater that $70\mu s$ and only after, the $receive$ primitive is executed, blocking the CPU during $50\mu s$. On a 300 MFLOPS processor such as the DEC 21064 Alpha, it is a long time spent on communications. The latency on the transmission is in the order of $2\mu s$. Howell observed similar numbers for MPI on the SP2 machine.

The J-machine: Designed by Bill Dally and developed by MIT in conjunction with Intel Corporation, the J-machine is a fine-grained distributed memory computer. The machine was built in VLSI technology in order to give it low cost and effectiveness. The design objective was the construction of a scalable machine, with independent and cheap processing elements and a fast communication interconnection system, and therefore, is classified as a distributed memory multicomputer. In order to fulfil these expectations, the processors are closely integrated with the network interface, so that there is a low overhead on the communication of messages. The topology is a mesh and a wormhole mechan-

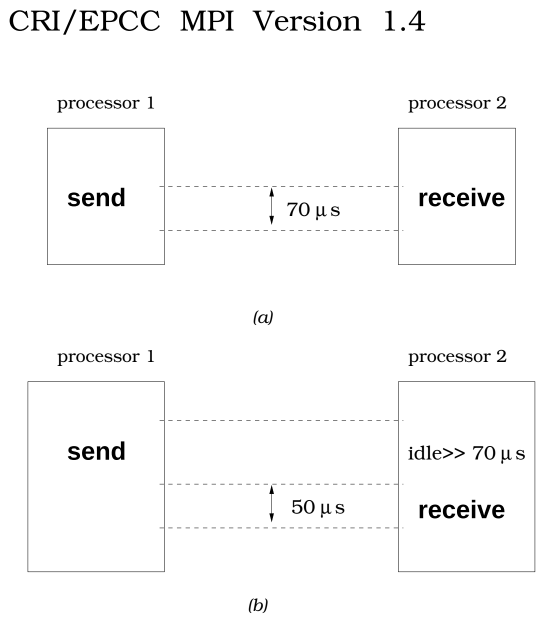


Figure 2.3: The sending and receiving of messages using CRAY T3D MPI.

ism is used to route messages between pairs of processors.

Each processor is associated with an on-chip memory and an off-chip memory (the latter was added to overcome a design limitation with the original local memory). Buffers are used for the receipt of messages by the on-chip memory to minimise interference with processing and instruction fetching. The hardware supports a synchronisation mechanism.

There exists a high degree of integration between the processors and the communication mechanism in the J-machine; the processors being message-driven. One of the main features resulting from this integration is that the communication latency is hidden by process suspension, which is carried out without copying overheads. Also, the message dispatch is fully implemented in hardware.

The J-machine, rather than incorporating one parallel programming model, includes primitive mechanisms for efficient synchronisation and communication. The machine is being used as a platform for software experiments in fine-grained parallel programming. Software issues like load-balancing, scheduling and locality have yet to be exploited and specified. This research machine have focussed on optimising the processor network interface. Although a significant improvement over the commercial machines, the startup overhead still predominates on the communication cost, as illustrated by measurements reproduced from [34] below. In Table 2.3 O_{send} and O_{recv} are the overhead to send and receive a message which are incurred on the communicating processors, correspondingly, r is the delay through each intermediate processor and is the channel width:

Network	Cycle(ns)	w(bits)	$O_{send} + O_{recv}$ (cycles)	r (cycles)
3d Mesh	31	8	16	2

Table 2.3: Communication measurements for a one-way message without contention.

2.6 Network of workstations

One of the most claimed problem with global memory multiprocessors is the difficulty of increasing the number of processors and the memory when scaling up becomes a necessity. Many researchers are interested in the development of distributed memory machines as they are scalable and it is a commonly observed phenomenon that as time goes by, users discover the need to run larger applications.

Networks of workstations have become very popular: a set of very fast VLSI technology processors, easily accessible by many users and cheap enough so that many of them can be easily acquired. Parallel computers consists of a set of processors with workstation-like technology and a communication system that is more efficient than the existing ones adopted by networks of workstations. However, parallel computers are much more expensive and when using a network of workstation the user has the choice of accessing either a single fast processor or many processors, which do not have to be homogeneous.

Among many researchers [8, 34, 49, 94] there exists a strong feeling that computation is converging towards local area networks and massively parallel computation will be processed by networks of workstations. The rate of development of workstation technology is very fast, if compared with the development rate of existing supercomputers. The ratio of price to performance in workstations improves at 80%, while that of supercomputers at 20% to 30% per year.

It is well known that with the developments in VLSI, processors are becoming faster. Also, both DRAM memories and disks are improving at a very high rate [110], although still slower than the improvement rate of CPU technologies. However, there are many aspects that still have to be resolved when considering a network of workstations, which are related to the communication system. In MPP systems, *latency* (as defined in the BSP and LogP models) is an important feature, although there are systems in which it can be overlapped with computation, as it does not require any work from the processors. Another relevant characteristic is the *overhead* spent by a processor when preparing a message to be sent or received; this is a *blocking* parameter, as it is the processor's job to pack or unpack the message.

The evolution of technology corresponds to the effort spent by researchers in order to solve such communication problems, so that, efficient parallel systems are developed. In the early times high performance microprocessors with their caches were connected via a single bus. If a small number of processing elements are connected, cache could experience lower traffic by keeping consistency with global memory through the bus. However, a greater number of processors is required. The problem is the bus traffic due to cache consistency. As a solution, data is replicated on the caches for a faster access and minimisation of the communication cost. As a result, *cache coherency* protocols are implemented, and are also used as a method of synchronising processors. Still, cache coherency and processor synchronisation using a bus as a link between processors and global memory have many implications which leads to a limit on the number of processors that is not very attractive [111]. It is difficult to combine characteristics like high bandwidth, low latency and unlimited bus length when using a network of workstations.

Although the Ethernet is used to connect workstations, it consumes too much bandwidth [33]. It is a simple or branching bus-like network and even when idle, the time to fetch data is very slow. On a Sun SPARC Station-10, the startup overhead added to the network latency incurred when sending a single message on an unloaded system can be up to $456\mu s$, with 9Mb/s of bandwidth. If the Ethernet is substituted by an ATM, the bandwidth goes up to 78Mb/s, but the communication time increases to $626\mu s$ [109].

When executing a parallel application, the processors should be able to execute tasks very fast, the network should be scalable and the values for latency and overhead when communication is carried out should be very low. Although the components within a network of workstations are very efficient devices when considered separately, the interface between the components can be a bottleneck when communicating.

The solution found by researchers for the problem of high traffic when using a single bus was the development of interconnection network. Instead of being used for every memory access as in a bus configured system, an interconnection network is accessed only when interprocess communication is incurred.

In message passing systems, machines are designed to have their dimension increased to thousands of processors. It is well known that topologies are not uniform among different systems. For example, the nCUBE/2, is a multicomputer with a few thousand nodes connected in a hypercube network [8]. Another example, is the CM-5, also, with a few thousand SPARC RISC processors connected in an incomplete fat-tree [84, 85]. In Table 2.4, the incurred costs in some message passing systems are shown. The startup overhead is represented by λ and the per-byte cost is denoted by τ. The notation $\mathcal{W}$ represents the average cost of a floating-point operation. It can be observed that the startup overhead is huge when compared with the computation time. The measurements can be found in [37, 38, 39, 67].

Machine	λ μs/msg	τ μs/byte	$\mathcal{W}$ μs/flop
iPSC	4100	2.8	25
nCUBE/10	400	2.6	8.3
iPSC/2	700	0.36	3.4
	390†	0.2	
nCUBE/2	160	0.45	0.50
iPSC/860	160	0.36	0.033
	60†	0.50	
CM-5 ‡	86	0.12	0.33

†: messages up to 100 bytes
‡ blocking send/receive

Table 2.4: Some Measurements on Message Passing Systems: overhead (λ), latency (τ), computation cost ($\mathcal{W}$)

2.7 Conclusion

Not only does the exploitation of locality inherent in applications need to be addressed when designing portable parallel programs, but so does the startup overhead. Indeed we would argue that startup overhead is sufficiently important to be incorporated explicitly in the computation model. There are many parallel systems that pipe messages using the *wormhole routing* mechanism, in order to hide the network latency. However, the startup overhead of the protocols designed to pipeline messages can be very high. Furthermore, the overhead is influenced by calls to the operating system and by the interfaces between the processors and the interconnection network [73]. Although measures are being taken so that communications incur smaller overheads than the existing ones, it is most likely that the startup overhead will continue to have a significant impact on the communication cost. With high startup costs, sending small messages to a neighbouring processor may have the same effect as sending large messages, and therefore, it may be wise to group many small messages into fewer but larger ones.

One can conclude from the diversification of models of parallel computation that there are many issues yet to be decided and resolved. For example, the question of whether a model should be synchronous or asynchronous is still being discussed. To answer questions like this, one should observe the trends for the future by analysing how current parallel computers are being designed. One should bear in mind that most parallel machines are asynchronous [111] and as discussed in Section 2.2, synchronous models can be simulated efficiently on asynchronous machines.

Due to the extensive use of workstations, many researchers have foreseen that this class of machines is the choice for the majority of both scientific and non-scientific users. Therefore, there exists a strong motivation for the development of parallel machines based on processors of an equivalent technology. But then, an issue that comes to light is whether these processors should share a global memory or the memory should be physically distributed. The demand for a fast response system for very large applications reassures the community that massively parallel processing is a necessity and therefore, memory should be physically distributed.

A massively parallel processing system is defined as an architecture in which hundreds of inexpensive and fast processors (workstation-like) are set to work simultaneously on complex problems. Each processors has its own local memory, making it easier to scale up the number of processing elements (PEs). The PEs are connected using interconnection networks under some topology efficient enough to allow the scaling up of processors without harming the overall performance of the system. These systems are likely to support coarse-grained applications, because the communication overhead is significant when parallelising applications. In the case of fine-grained applications which are generally communication intensive, this will lead to poor performance in such systems. Therefore, techniques to exploit locality and to appropriately allocate the application onto the parallel machine must be provided.

One trend for the future is that each "processing element" will correspond to a *small multiprocessor*, i.e. a small set of processing elements communicating via a physical shared memory. These *small multiprocessors* are connected using an interconnection network of some specific topology, as in a distributed memory multicomputer, thus defining a *hybrid* system that supports a hierarchical memory system. At the first level of the hierarchy which corresponds to the *small multiprocessor*, fine-grained applications could be performed reasonable well with the second level supporting coarse-grained applications. This idea "models" hierarchical parallelism which may be very powerful for the exploiting the hierarchical nature of parallelism found in a variety of applications.

Some new machines machines already implement this hybrid system, e.g. the Convex SPP1200 system [21]. However, this particular machine emulates the global memory concept on a physically distributed memory, claiming that portability of existing programming environments will not be a problem. Such systems support shared memory and message-passing mechanisms. Also, the emulation of a shared memory is supposed to promote low communication costs. One relevant point is that the current SPP1200 has at most 128 processors. In massively parallel processing this is not a very high number and although it is claimed that an efficient "virtual" shared memory machine has been implemented, locality still needs to be exploited well [10, 11].

Although the supercomputer manufactures claim their communication systems are efficient, the interface between the processing elements and communication systems has yet to be taken into account. In a network of workstations, the interface between processing elements and the communication system is of a more critical concern. Even today the costs of supercomputers still exceed the sum of the costs of their components and are more expensive than an equivalent network of workstations given that they are both based on the same processor technology. Added to the low cost, the practicality and independence provided by networks of workstations for their users are issues which explain the high acceptance of such a class of parallel machines.

Concluding the above discussion, the characteristics that need to be incorporated into any standard model of parallel computation if a wide range of machines is going to be represented, are:

- parameters related to communication:
  - the startup overhead;
  - the latency on the communication between adjacent processors;
- memory is physically distributed among the processors;
- enough details of the interconnection network so that locality within the application may be exploited;

- asynchronous communications so that the communication latency can be overlapped with computation (no global synchronisation needed).

Specifically, these essential characteristics must to be taken into account in scheduling strategies in order to exploit well the two most important issues in parallel computation: locality and minimisation of communication costs.

Chapter 3

On the Definition of Scheduling Models

3.1 Introduction

Many researchers have been attempting to propose a realistic model of parallel computation, but from the classes of parallel architectures and the models of computation discussed in Chapter 2, it is difficult to point to a model that specifies all of the important issues related to the scheduling of an application on a parallel machine. However, from the discussion, important characteristics of parallel computation were identified. Our main objective is to schedule an application onto a given target architecture in the best way possible, and for doing so, characteristics of a wide range of applications and architectures must be well defined. However, only the most important characteristics should be parameterised, because a very detailed model definition is difficult to use as a framework for schedule algorithms.

3.1.1 The architecture model

It is agreed by many researchers [140] that when the number of processors tends to be very large, memory should be distributed. Also, algorithms devised for parallel machines with global memory can be emulated reasonably efficiently on a distributed memory model [49, 66, 72]. The architecture model defined here is based on the distributed memory model and should have enough information to represent the given target architecture. These assumptions lead to the following characteristics of the model to be present in this thesis:

- a set of m processing elements, each of which consists of a processor and its local memory. The set of processors is denoted by $\mathcal{P} = \{p_0, p_1, \ldots, p_{m-1}\}$

and the processing element corresponding to the processor p_i is denoted as PE_i;

- the time that a processor takes to prepare a message to be sent or received, called *overhead*, is denoted by λ_s and λ_r, respectively;
- the time elapsed from when the first bit of data (in units of data) passes onto a link between two processors until the last bit of data leaves the link is represented by an integer τ, called as *delay* or *latency*.

The overhead for sending and receiving are denoted by λ_s and λ_r respectively. However, whenever the overhead is denoted by λ, it is assumed that both quantities have the same value.

It is not always the case that all processors are connected to each other. An interconnection network with a specific topology, used to connect the processors, is represented by an undirected graph $G_{\mathcal{A}} = (V_{\mathcal{A}}, E_{\mathcal{A}})$ where the set of vertices $V_{\mathcal{A}}$ represents the set of processors $\mathcal{P}$ and the set of edges $E_{\mathcal{A}}$ corresponds to the bidirectional links existing between adjacent processors. One of the reasons processor topologies are represented by graphs is that the existing knowledge of graph theory can be exploited. For example, there are algorithms which define the shortest path between two vertices, searching algorithms used for routing techniques and characteristics like out degree and in degree of each vertex [14].

The minimum number of edges between two processors (not necessarily adjacent) is the number of hops between them, which is represented by the function $h : \mathcal{P} \times \mathcal{P} \to Z_0^+$. When a data unit is transmitted between two adjacent processors, that is, two processors connected by just one link, the communication delay, i.e. latency, is τ. Therefore, the latency associated with the sending of a data unit between any two distinct processors $p_i, p_j \in \mathcal{P}$ is $\tau * h(p_i, p_j)$ (i.e. it is a contention free model), and it is denoted by the function $\varphi : \mathcal{P} \times \mathcal{P} \to Z_0^+$.

The communication carried out between processors is dictated by the structure of the input application and its schedule: when tasks allocated to different processors need to communicate, the source processor sends data to the destination processor. The processors communicate in an asynchronous mode, in the sense that when a message is already on a link, the communicating processors can execute other tasks. Unlike the BSP model-style [128], there is no global synchronisation.

3.1.2 The application model

An application is a set of tasks which may interact with each other through the exchange of messages. The class of applications addressed has the following characteristic: each task can only be executed whenever all necessary data (received through input messages) is locally available. After a task completes execution, the resulting data is sent through output messages, if necessary. The characteristics of each task remain throughout the execution of the application without any dynamic creation of other tasks. This class of applications is represented by a static directed acyclic graph (*DAG*) $G = (V, E)$, in which V is a set of n vertices $\{v_1, v_2, \ldots, v_n\}$ corresponding to the tasks and E, the set of edges representing the precedence relationship among tasks, i.e. $E = \{(v_i, v_j) : v_i, v_j \in V\}$. In this dissertation, the terms task(s) and vertex(vertices) are used interchangeably. The set of immediate predecessors of a task $v_i \in V$, denoted by $pred(v_i)$ and the set of immediate successors, denoted by $succ(v_i)$, are such that

$$pred(v_i) = \{v_l \in V : (v_l, v_i) \in E\}$$

$$succ(v_i) = \{v_j \in V : (v_i, v_j) \in E\}$$

Other characteristics may be associated with an application, providing a greater knowledge about it and also opportunity to classify classes of *DAG*s. The number of computation steps necessary to execute a task defines its execution cost. The execution cost is expressed in units of measurement that reflect the relative duration of that task. Each computation step takes one *time unit*, but due to the different speeds of various machines the absolute duration of a *time unit* is machine-dependent. The execution cost of a task $v_i \in V$ is denoted by the function

$$m : V \longrightarrow Z_1^+$$

Therefore, $m(v_i)$ is the number of the time units incurred due to the execution of task v_i on any processor of the network.

Weight may be associated with an edge $(v_i, v_j) \in E$ representing the amount of data transmitted from v_i to v_j. The weights are denoted by the function

$$w : E \longrightarrow Z_1^+$$

i.e. $w((v_i, v_j))$ is the number of data units sent from v_i to v_j. The edge between two tasks can have different interpretations. It can simply represent the precedence between the tasks, and no data is being sent, but just a data control.

Non-unit weights associated with the edges represent not just the precedence relation, but also the transmission of data. Therefore, when it is mentioned that "data is being communicated" between two tasks, it is implicit in this context that it is data control or a certain amount of data that is being communicated.

We now define some important characteristics of *graphs* which are used throughout this dissertation, due to their usefulness in approaches and strategies yet to be discussed. The concept of *paths* in a *DAG* and their length, and also, the order of tasks implicitly defined in a *DAG* are some of these important characteristics.

Definition 3.1.1 *Given a DAG* $G = (V, E)$*, the path between any two tasks* $v_{i_1}, v_{i_k} \in V$*, denoted by* $path(v_{i_1}, v_{i_k})$*, is the list of edges* $\langle(v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_k})\rangle$ *such that each edge* $(v_{i_l}, v_{i_{l+1}})$ *is in* E*:*

$$path(v_{i_1}, v_{i_k}) \;=\; \langle(v_{i_1}, v_{i_2}), \ldots, (v_{i_{k-1}}, v_{i_k})\rangle \;:\; (v_{i_l}, v_{i_{l+1}}) \in E,\; l = 1, \ldots k-1$$

The path $path(v_{i_1}, v_{i_k})$ *is of length* k*.*

The shortest path between two tasks $v_{i_1}, v_{i_k} \in V$ *is the smallest path among all paths between* v_{i_1} *and* v_{i_k} *and it is denoted by* $spath(v_{i_1}, v_{i_k})$*. In the case when there are several paths of the same length, ties are broken arbitrarily.*

Definition 3.1.2 *Given a DAG* $G = (V, E)$*, the topological order of a task* $v_i \in V$*, denoted by* $\mathcal{T}(v_i)$ *is the length of the shortest path between the source task and* v_i*, i.e.*

$$\mathcal{T}(v_i) \;=\; |\; spath(v, v_i) \;|$$

where v *represents the source task.*

3.2 Communication events and schedule

When sending or receiving messages, a processor spends some time "preparing" the message to be sent or received. During the time in which a *communication event* occurs, the processor is *blocked*, not being able to execute other tasks in V or even to send or receive other messages. Taking into account these characteristics, when an application is executed on a given parallel architecture, a processor $p_i \in \mathcal{P}$ can be in one of the following states: computing a task, sending a message, receiving data and idle.

The sending and receiving events are also "tasks" executed by the processors, which are represented by a set of vertices $\mathcal{E}$. In order to formally define this set, we need to know certain characteristics of the schedule of tasks. Throughout this dissertation, we shall see that we consider schedules that incorporate task replication and also, schedules in which tasks are not replicated. In order to represent the processor(s) allocated to a task we define the following necessary notation.

Definition 3.2.1 *Let v_i be a task. On a schedule without task replication, v_i is allocated to a single processor denoted by $proc(v_i)$. On a schedule that allows task replication, the list of processors to which v_i has been assigned is denoted by $processors(v_i)$.*

Suppose $p, p' \in \mathcal{P}$ are two distinct processors which execute v_i and v_j respectively, given that $(v_i, v_j) \in E$ and $p \notin processors(v_j)$. The communication event carried out on processor p when sending a message from v_i to v_j is represented by the *sending event* task $\sigma_{(v_i,v_j,p)} \in \mathcal{E}$. The task executed by processor p' associated with the receiving of such message is the *receiving event* task $\rho_{(v_i,v_j,p')} \in \mathcal{E}$. Formally,

$$\mathcal{E} = \{\sigma_{(v_i,v_j,p)}, \rho_{(v_i,v_j,p')} : (v_i, v_j) \in E \ \wedge \ p \notin processors(v_j)\}$$

3.2.1 Valid schedule of a *DAG* on the target architecture

When executing an application in parallel on the target machine, the tasks are executed according to their schedules, considering characteristics of the application, like the precedence relationship, and also characteristics of the target architecture, like communication issues already discussed. The schedule time of a task $v_i \in V$ corresponds to the time that v_i starts to be executed on a processor $p_i \in \mathcal{P}$. It is important to address when and where a task is executed.

The schedule times and allocation of the tasks must be consistent with a set of conditions, so that a *valid* schedule is provided. When defining this set of conditions, the reader should bear in mind that we do not require an algorithm to determine whether some structure is a valid schedule, rather we need to define precisely the characteristics that a given *heuristic* should embody when producing a schedule for the application.

Definition 3.2.2 *Let the function $s : V \times \mathcal{P} \longrightarrow Z_0^+$ characterise the time at which a task v_i starts being executed on a processor p_j. The task v_i finishes execution at time $s(v_i, p_j) + m(v_i)$, which is represented by the function $f : V \times \mathcal{P} \longrightarrow Z_1^+$.*

A schedule S is a finite set of tuples $(v, p_j, t_k) \in V \cup \mathcal{E} \times \mathcal{P} \times Z_0^+$ which satisfies the following set Γ of necessary conditions:

1. Each task $v_i \in V$ is executed on some processor. Therefore, for every task there exists at least one tuple in S. Formally,

$$\forall v_i \in V, \;\; \exists (v_i, p_j, t_k) \in S$$

and therefore, $\forall v_i \in V, \;\; processors(v_i) \neq \emptyset$.

2. Each processor executes only one task at any one time. For all $p_j \in \mathcal{P}$, if there exists a task $v \in V \cup \mathcal{E}$ such that $(v, p_j, t_k) \in S$, there is no other task $u \in V \cup \mathcal{E}$ such that $(u, p_j, t_y) \in S$ and $t_k \leq t_y < t_k + e$, where either $e = m(v)$ if $v \in V$, or $e = \lambda_s$ or $e = \lambda_r$, if $v \in \mathcal{E}$;

3. A task v_i can be executed only after all its immediate predecessors are completed and all respective necessary data is received. More formally, a tuple $(v_i, p_j, t_k) \in S$ is such that

$$\forall v_l \in pred(v_i), \;\; \exists (v_l, p_y, t_w) \in S \;\; \text{such that } t_k \geq t_w + m(v_l) + \alpha$$

where α corresponds to the communication cost incurred due to the sending of the message (v_l, v_i).

4. Let S_{p_j} be the subset of tuples $(v_i, p_j, t_k) \in S$ corresponding to the execution of all tasks on the processor $p_j \in \mathcal{P}$, such that

$$S = \bigcup_{p_j \in \mathcal{P}} S_{p_j}$$

S'_{p_j} is a permutation of S_{p_j} such that the elements appear in increasing order of the schedule times of the tasks allocated to p_j:

$$S'_{p_j} = \langle (v_i, p_j, t_k) \mid t_k < t_{k+1} \rangle$$

where the angle brackets denote an ordered set.

The following S_{p_j} properties are:

(a) $t_0 \geq 0$

(b) The processor p_j is in one of the following states: executing a task, preparing to send/receive a message or idle. Therefore, supposing that $(v, p_j, t_k) \in S_{p_j}$, in which $v \in V \cup \mathcal{E}$, the schedule time t_{k+1} is such that

$$t_{k+1} \geq \begin{cases} t_k + m(v_i) & \text{if computing} \\ t_k + \lambda_s & \text{if sending} \\ t_k + \lambda_r & \text{if receiving} \end{cases}$$

(c) A *sending* event task is scheduled whenever there exists a task, say, v_i, mapped to a processor, say, p_y at time t_x so that $(v_i, p_y, t_x) \in S_{p_y}$ and p_y is such that for some $v_l \in pred(v_i)$, $p_y \notin processors(v_l)$. Then, $\exists p_j \in processors(v_l)$ and $\exists t_k$ such that $t_x \geq t_k + \lambda_s + \alpha,\ \ \alpha$ denoting the cost associated with the communication between p_j and p_y, and $\exists t_w$ such that $t_w + m(v_l) \leq t_k$, where

$$(\sigma_{(v_l,v_i,p_j)}, p_j, t_k) \in S_{p_j}$$

and

$$(v_l, p_j, t_w) \in S_{p_j}$$

(d) A *receiving* event task $\rho_{(v_l,v_i,p_y)}$ is scheduled on p_y whenever there exists *sending* event tasks $\sigma_{(v_l,v_i,p_j)}$ allocated to a distinct processor. The existence of both tasks is due to the execution of the task v_i on p_y which has an immediate predecessor v_l not scheduled on the same processor. The time at which the *receiving* event task is scheduled is after reception of the respective message. Formalising, if $(\rho_{(v_l,v_i,p_y)}, p_y, t_x) \in S_{p_y}$ with $(v_l, v_i) \in E$:

$$\exists(\sigma_{(v_l,v_i,p_j),p_j,t_k)} \in S_{p_j}, \quad p_y \neq p_j$$

such that

$$t_x \geq t_k + \lambda_s + \varphi(p_y, p_j) \cdot w((v_l, v_i)$$

If all the conditions in the set Γ are satisfied, the schedule S is considered a *valid* schedule. The *makespan* of S, denoted by $\mathcal{M}(S)$, is the time to completion of S, i.e

$$\mathcal{M}(S) = \max_{v_i \in V}\{f(v_i)\}$$

As a matter of notation, whenever it is necessary to indicate that S is the set of schedule tuples of a *DAG* G, $S(G)$ is used to provide a clearer and more precise notation.

3.2.2 The CLAUD model

The *Cost and Latency Augmented DAG* (CLAUD) model suggested in [102] and established as a scheduling model in [24] incorporates the relevant characteristics discussed in the definition of the Architectural model and Application model. These characteristics are considered essential to describe properly a given application and the target architecture, when the aim is to design a schedule strategy for a wide range of applications and parallel machines. One of the main characteristics tackled in the CLAUD model is the existence of *communication events* as tasks, taking a special care about them when devising scheduling strategies. In this dissertation the CLAUD model is adopted as the target schedule model.

In the literature, one can find many scheduling approaches in which the overhead is incorporated in their respective models (e.g. [46, 136, 87, 126, 48], but they incorporate the overhead in the latency cost. Throughout this dissertation, it will be concluded that this is not always the best approach. In conclusion, the CLAUD model has the following characteristics:

Application: A *DAG* $G = (V, E)$ represents the application, which is a set of tasks related to each other;

Processors: a set $\mathcal{P}$ of m identical processors;

Overhead: when communicating there is a penalty incurred in each communicating processor in the form of a class of tasks, the communication events. The overhead when *sending* or *receiving* a message is λ_s and λ_r respectively;

communication events: A communication event task must be scheduled by each pair of sending and receiving processors. Let $(v_i, v_j) \in E$. For all $p \in processors(v_j) - processors(v_i)$, $\exists p' \in processors(v_i)$ such that the *sending event* tasks and *receiving event* tasks are specified in the set

$$\mathcal{E} = \{\sigma_{(v_i, v_j, p')}, \rho_{(v_i, v_j, p)} : (v_i, v_j) \in E\}$$

Topology and Delay: the latency incurred when two processors p_i and p_j communicate is $\varphi(p_i, p_j) = \tau * h(p_i, p_j)$, in which τ is the delay incurred when two adjacency processors communicate and $h(p_i, p_j)$ is the number of hops between p_i and p_j.

3.3 A survey on scheduling heuristics

The problem of scheduling an application, usually represented by a *DAG*, on a parallel machine so that an optimal makespan is achieved is known to be NP-complete [127] in its general form. Many heuristics which produce approximated optimal solutions, have been proposed in order to produce a "good" schedule for the input *DAG*. There is no guarantee that heuristics produce optimal solutions for any *DAG* onto any system, but many of these heuristics may find near-optimal solutions which are within some factor of the optimal.

The scheduling field is a wide area, where the various scheduling problems being tackled may have many different characteristics. A scheduling strategy is proposed under certain assumptions which are related to the given application and target architecture. In this section various scheduling schemes are investigated and briefly described in order to discuss the many techniques and scheduling models already proposed in the literature.

The taxonomy for scheduling approaches is found in the literature such as [22, 26, 42, 91, 127]. It is assumed that the information of the input application does not change with time and also, this information is available before the execution of the scheduler. Therefore, a technique used to schedule such a *DAG* is classified as *deterministic* or *static scheduling*. The scheduling approaches which are described are *non-preemptive*, therefore, tasks cannot be interrupted once their execution has started.

Different scheduling techniques were proposed to tackle different scheduling models, although many of them are similar, where architectural characteristics are specified. In the literature, and also in our target architecture model defined in Section 3.1.1, two communication parameters are identified as being very important: the delay on the communication and a form of *CPU penalty*, which is incurred on the communicating processors. However, it is not always the case that both characteristics are taken into account by a scheduling approach, since, depending on the target machine, one of these characteristics is considered negligible. We propose a classification of the wide range of scheduling techniques (Figure 3.1), dependent on how the communication parameters are taken into account:

- the delay is the only communication parameter considered;
- an abstract *CPU penalty* form;
- *CPU penalty* and delay are the main communication parameters.

In the following sections, a discussion in the scheduling strategies in each class is presented.

3.3.1 Communication cost modelled as delay

With the objective to determine the trade-off between the degree of parallelism and communication costs, many scheduling heuristics were proposed consid-

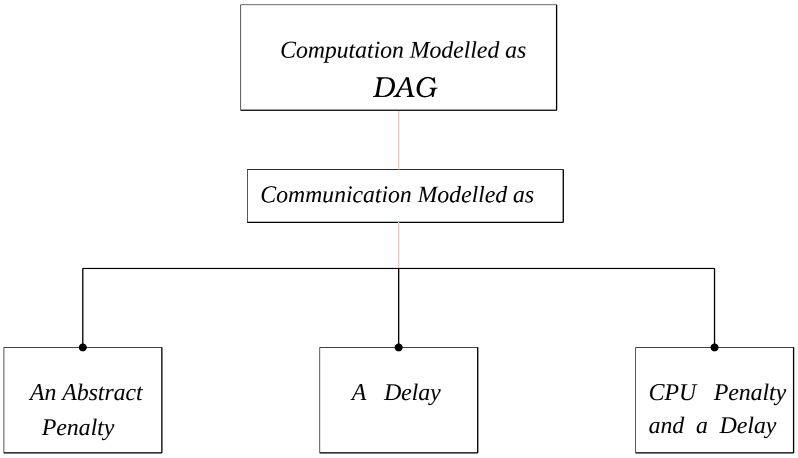


Figure 3.1: Classification of the scheduling strategies

.

ering delay as the sole and most representative parameter of the architecture. Usually, the communication between adjacent tasks allocated to the same processor is negligible, since the cost of communication between distinct processors is much higher than the speed of a processor.

The scheduling techniques in this category assume that task execution can be overlapped with the communication between processors. Messages may be sent and received in parallel and usually, processors are able to multicast data to various processors and receive data practically at the same time. An asynchronous execution model is specified, where communication and synchronisation are implemented explicitly. Usually, for the same amount of data, the communication takes a uniform time. This message passing model is adopted by, for example, Yang *et al.* [46], Yu [139], Kruatrachue and Lewis [78] and Sarkar [116] and others [41, 46, 65, 88, 106].

In many scheduling heuristic proposals, a weighted *DAG* specifies both application and architectural model. Each task of the *DAG* is associated with an integer which represents the execution cost of the task [45]. The weight associated with each edge, represents the *communication cost*, which incorporates not only the message size but also the delay to send the data. These scheduling heuristics are usually clustering techniques that partition the *DAG*. Usually, subsequent stages are implemented so that the actual mapping of the clusters onto processors is carried out.

In [9, 45, 78] the communication cost is a linear model, where the time spent to send a message of size w is $\alpha + w\beta$, given that α is the startup overhead and β, the transmission rate. The manner in which the startup cost is modelled shows clearly that it is not referred as a *CPU penalty*, as it is specified in models like CLAUD [24] and LogP [34], but in this case, the *channel processors* are independent of the *task processors* [9]. Also, the assumption that each edge is linearly modelled leads to the conclusion that the processors are fully connected.

Kim and Browne specify, in [75], a weighted *DAG* as representation of the input application and another weight *DAG* as representation of the target architecture. A pair of integers is associated with each vertex-processor, representing the processor power and the amount of memory associated with node. Also, a function maps each edge-channel to an integer, representing the bandwidth of communication channel.

McCreary and Gill [97] specified the linear model as in [78], but also, a task receives all of its messages sequentially from its immediate predecessors, and only after its completion, communicate is carried out. The so-called *additive communication* reflects the assumption that there exists only a single channel between the communicating processors. Polychronopoulus also assumed in [113] the *additive* model considering the serial communication being done via bus connected to a shared memory.

Hwang *et al.* in [9] and [65] proposed a communication model in which each edge of a *DAG* is associated with the amount of "messages units" (packets, for example) to be transmitted between the adjacent tasks. Separately, a parameter is defined in order to represent the time required to send each message unit between to distinct processors, which incorporates the distance between them. The delay associated with each channel is constant. These assumptions are also considered by Veltman *et al.* [130].

Taking into account different techniques used by the many scheduling applications, we try to classify these approaches into groups, in a similar manner to that found in [42, 74]. We now describe and discuss some representative approaches in each group.

3.3.1.1 List Scheduling Strategies

The list scheduling is a heuristic method to schedule *DAG* onto a set of processors by assigning priorities to the tasks. Whenever there exists an *idle* processor, the heuristic schedules an *available* task with the highest associated priority. A task is *available* when all its immediate predecessors have already been scheduled. The main difference between the many list scheduling algorithms proposed in the literature is the priority used to select the next task to be scheduled, and also, certain characteristics of the computation and communication models. A framework for this class of heuristics can be formalised, as described below:

Framework 1 *List Scheduling* {

```
        Define a priority to be associated with each task of the input DAG;
        while there are tasks to be scheduled do {
            select an available task with the highest priority;
            select an idle processor on which the task starts earliest;
            schedule the task on the selected processor;
            set the new available tasks;
        }
    }
```

A list scheduling heuristic can be seen as a sequence of events: tasks are scheduled whenever they become *available* and whenever there exists an idle processor in which the task can start the earliest. Variations of the *list scheduling method* incorporate a *clock* into their algorithm, which controls the order of these events. Usually, the strategy has one stage, in which decisions are made iteratively and depend on the current *state of the system*. The start time of a task depends on the schedule and allocation of its immediate predecessors. The list scheduling strategies are non-backtracking in order to avoid an increase in the complexity of the scheduling algorithm. The nature of such a strategy may lead to a solution that is a local minimum, rather than global.

In an attempt to unify the terminology used in the list scheduling field, we specify some useful and necessary notations. Given a *DAG* $G = (V, E)$, the *level* of a task $v \in V$ is the longest path from v to the sink task (if there exist more than one sink, a *dummy* sink is introduced as their immediate successor). The *co-level* of v is the longest path from the source task (in the same manner, a *dummy* source task is defined if there is more than one source) to v. The concept of *level* and *co-level* come from graph theory since when first adopted as priorities in list scheduling strategies, unit execution costs were associated with the tasks and no communication costs were assumed. Also, *critical path* analysis corresponds to list scheduling technique with *level* priority. Coffman and Graham [27] first adopted *level* of tasks as priority in their proposed list scheduling heuristic and showed that when this priority is used, the list scheduling produces the optimal result considering tasks with the same execution cost and two processors.

When first studying the scheduling problem using list scheduling algorithms, heuristics were proposed without considering communication delays. The avoidance of communication costs may be accepted in shared memory computers where messages are passed at memory cycle speeds and also, for the sake of the analysis of new techniques. Since execution costs are considered, they are incorporated when calculating the *level* or *co-level* of the tasks. Hu [64] introduced the list scheduling with *level* priority, no communication delays and with non-unit execution costs associated with the tasks. With these same costs assumptions, Pathak [108] introduced the *heuristic* algorithm which uses the *co-level* priority. Adam *et al.*, in [1], evaluate the behaviour on different versions of list scheduling using *level*, *co-level* and a *random* priority, respectively. They experimentally concluded that when using the *level* priority for non-unit tasks and no communication costs, near-optimal schedules are produced. The *weighted length* algorithm introduced by Shirazi *et al.* [119] is an improvement on the list scheduling. The priority assigned to each task depends not only on its *level*, but also on the number of immediate successors, the weight of the successors and the *level* priority of the successors. The definition of these many characteristics as priorities is an attempt to incorporate information beyond the adjacency of the tasks in order to avoid local solutions.

The incorporation of the communication delays correspond to the fact that, when calculating the *levels* or *co-levels*, the weights associated with the edges are considered. However, as discussed by Rewini [42] and Yang [135], these priorities do not remain the same during the scheduling process, because the communication cost varies depending on the allocation of the tasks. The list scheduling framework is implemented with communication costs incorporated. Yu proposed, in [139], a modification to Hu's list scheduling strategy including communication costs.

The *Insertion Scheduling heuristic* (ISH) proposed by Kruatrachue *et al.* [42, 77, 78], uses the *level* priority and assumes non-unit execution of tasks and communication costs. The heuristic tries to improve the schedule through the assignment of *available* tasks on *idle slots* of processors. The benefit produced by this strategy can be observed when there are many processors, since in this case, many idle slots may be defined. However, this approach does not fully exploit the degree of parallelism, due to the local decisions which are carried out.

In the above strategies where communication is considered, a delay is incurred when the communicating tasks are allocated to different processors and therefore, decisions are taken on whether adjacent tasks are allocated to the same or distinct processors. Hwang *et al.* in [65] proposed a more realistic scheduling algorithm which considers delay on communication and the notion of network topology. In the *Earliest Task First* algorithm (ETF), execution costs and weights are associated with the tasks and edges of the input *DAG* respectively. An *available* task is allocated to an idle processor where it will have the earliest start. The scheduling process is guided by a clock and, if an *available* task can start earlier on a current *busy* processor, it waits for the processor to become idle. Although more realistic architectural characteristics are incorporated, the schedule produced by ETF is data dependent.

In a similar way to ETF, the *mapping heuristic* (MH), proposed by El-Rewini and Lewis [40, 41], considers several realistic issues like interconnection topology and delay and also, handles contention due to communication. An *available* task with the highest *level* priority (with communication costs included) and, if ties need still to be broken, those *available* tasks with the highest number of immediate successors are selected. In their scheduling model, the linear communication model is assumed, but startup overhead due to the sending of messages is not considered as a communication task. In order to compute the delay due to contention, MH maintains, for each processor, a *routing table* that contains routing information, indexed by all the other processors so that the best route to send a message is selected. Contention is a global characteristic of a parallel system and therefore, handling it through local decisions may lead to a local solution far from the optimal. The local optimality of list scheduling algorithms is shown by Yang *et al.* in [137]. Also, Sarkar and Hennessy [117] shows the difficulties of devising a single stage non-backtracking scheduling algorithm, for the delay model, which handles communication costs efficiently.

3.3.1.2 Clustering of Tasks

In this class of scheduling heuristics, priorities are also assigned to tasks which are clustered on each processor in order to minimise communication costs [46]. A cluster is defined as a set of tasks, while a task is an indivisible unit of computation, so that the tasks inside a cluster can communicate as soon as each has completed its execution, allowing the overlap of computation and communication. The main idea is to cluster tasks on the same processor in order to decrease the critical path length, since communication costs are being zeroed. Usually, the clustering strategies is part of a multi-stage scheduling strategy, since the clustering of tasks decides whether tasks are executed on the same processor or not. Then, it is necessary to map the clusters on the actual physical processors. These are characteristics that differentiates the clustering techniques from list scheduling techniques.

The clustering problem has been shown NP-complete in its many forms. Sarkar [116] and Papadimitriou and Yannakakis [106] have shown that the problem of clustering tasks in order to minimise the parallel time is NP-hard, even considering an unbounded number of processors.

Kim and Browne [75] defined the concept and proposed *linear clustering* where each one of the critical paths of a *DAG* is allocated to a processor. A *linear cluster* is one without independent tasks. Their algorithm *zeroes* communication edges of the *DAG* whenever the parallel time is decreased. After the *linear clusters* are defined, if there are more clusters than the number of physical processors, clusters are combined, in accordance with the following mechanisms: two clusters are merged if one starts after the other or if one can be executed while the other is idle. Although the combining mechanism reduces the number of clusters, it does not necessarily provide as many new clusters as the number of processors in the target architecture and therefore, further mechanisms might be necessary.

Sarkar and Hennessy [117] proposed a two-stage scheme where in the first stage, a clustering partition technique is proposed and in the second stage a processor assignment algorithm maps each cluster to a physical processor. Their clustering strategy, called the *Internalisation Pre-Pass* Partitioning algorithm, initially assigns each task to a distinct *virtual* processor. The algorithm visits each edge, merging each pair of adjacent tasks whenever the *critical path* length is minimised. The *critical path* priority defined is a combination of a task *level* and *co-level*. When no more improvements on the *critical path* can be achieved, the clustering stage terminates [62]. In the second stage, the clusters are assigned to processors using a modified Priority List Scheduling algorithm from Graham [54].

In the two clustering techniques briefly described above, the zeroing of edges provides a way to cluster adjacent tasks which communicate intensively and also, computation and communication can totally overlap. Motivated by this, Yang and Gerasoulis proposed the *Dominant Sequence Clustering* algorithm (DSC) [137, 138]. The DSC algorithm produces a schedule for a given *DAG* onto a fully connected network where tasks are *clustered* onto the *virtual* processors. The weight associated with the edges incorporates not only the amount of data transmitted between communicating tasks but also the cost of sending this data. DSC implements a sequence of edge-zeroing steps with the objective of reducing the dominant sequence (DS), which is defined as the *critical path* of a scheduled or semi-scheduled *DAG*. At each step of the algorithm, the DS is re-evaluated [138] and an *available* task with the highest *parallel time* priority (the summation of the task *co-level* and *level*) is selected. However, a task is not scheduled immediately. Firstly, a *partial available* task, which is the one with at least one of its immediate predecessors yet to be scheduled, with highest *parallel time* priority is evaluated. If the schedule of a *available* task onto a candidate cluster increases the current *co-level* of a *partial available* task with highest priority, the *available* task is not allocated to that cluster, but to a new cluster. These mechanisms characterises DSC as a clustering technique with a more global view of the state of the semi-scheduled *DAG*. DSC is part of the PYRROS system [136] which also comprises a *merge* algorithm performed whenever the number of clusters produced by DSC is greater than the actual number of physical processors. Finally, for the actual mapping of the resulting clusters, PYRROS incorporates Bokhari's [19] mapping algorithm. In order to produce a more accurate prediction, given the clusters of tasks and their allocation, the communication costs are readjusted to incorporate the processor distance, using the linear model $d(\alpha + w\beta)$, where d is the distance in hops between the communicating processors. Again, although the overhead when sending a message is characterised, it is incorporated into the delay.

The *Modified Critical Path* algorithm, proposed by Wu and Gajski [133] is considered a clustering scheduling algorithm, since it also zeroes edges in order to decrease the parallel time. The priority used to select an *available* tasks is *level* priority. Significant similarities between MCP, a clustering algorithm, and ETF [65], a list scheduling algorithm, can be found. MCP schedules an *available* task that can start executing first. In the experiments evaluated by McCreary *et al.* [74] lead to the conclusion that MCP initially allocated all the sources of a *DAG* onto distinct processors. In the case of a tree, this decision of parallelising all the sources causes the heuristic to have an excessive amount of communication in later iterations.

The next group of heuristics are those which allow task replication. Replication can be very important and useful, mainly when communication costs are very high. Jung *et al.* [70] have already evaluated the benefits of replication of tasks. Basically, the idea is to replicate a task on the same processor as an immediate successor so that communication cost is decreased. Many techniques were proposed and we shall note their similarities with already known strategies like list scheduling and clustering.

3.3.1.3 Strategy Based on Tasks Replication

The techniques described here considers that computation and communication can be totally overlapped. The first strategy based on the duplication of tasks was proposed by Kruatrachue [42, 77, 78]. The *Duplication-Scheduling Heuristic* (DSH) is an improvement over their list scheduling strategy ISH, by duplicating tasks whenever communication delay can be minimised. Let v be an *available* task and p, a candidate processor. If the start time of v is greater that the *idle* time of p (the time in which p is not executing any task), then there is some ancestor of v which is *delaying* its start time due to the communication. A search for this task is carried out, starting from the *latest immediate predecessor* of v, say, u. If u is not allocated to p, the delay on the communication between u and v postpones v's execution. So, u is replicated onto p. On the other hand, if u is already assigned to p a depth-first search is carried out, checking the latest immediate predecessor of the tasks in the search path. As soon as a latest immediate predecessor u' of some task in the path, allocated to a processor other than p, is found, the search stops and the task u' is replicated onto p. Experiments were carried out in [77] comparing DSH with Yu's list scheduling [139] and ISH, and it was concluded that in general DSH produces a better schedule, mainly when the delay has high values. These experiments assure the fact that replication minimises communication and may provide great benefits.

Ranka and Chung [25] proposed the *Bottom-up-Top-down Duplication Heuristic* (BTDH) which is essentially an extension of DSH, but it keeps on duplicating immediate predecessors even if the start time is not minimised, duplicating as long as there is an idle time slot [5].

Papadimitriou and Yannakakis proposed their polynomial heuristic [106], which produces a schedule with makespan within a factor of two of the optimum. They consider an unbounded number of processors and each task is replicated together with some of its ancestors onto a distinct *virtual* processor, given the earliest start time priority. This strategy is discussed in details in later chapters.

In [35, 36], Darbha and Agrawal devised an algorithm called *Search and Duplication Based Scheduling* (SDBS), which schedules a *DAG* with variable-size tasks onto an unbounded number of fully connected processors. The optimal makespan is achieved for tasks with certain conditions. One of the conditions is that the maximal amount of computation executed by any task is greater than the minimal amount of communication between tasks. Starting from the sink tasks, SDBS schedules each task v on the same *virtual* processor of its immediate predecessor u so that the earliest start time of v can be achieved. Therefore, a replication tree is defined and pseudo-critical paths are scheduled to each *virtual* processor. In the same manner as [106], no mechanism to schedule the clusters onto physical processors is devised. The algorithm proposed by Chen *et al.* in [23] also used linear clustering with task replication, but it has a higher complexity than SDBS. The *Small Communication Times* (SCT) with task replication of Colin and Chrétienne [29] has much stricter assumptions than those in SDBS, where a coarse grain application is also considered.

Ahmad and Kwok proposed the *Critical Path Fast Duplication* (CPFD) algorithm in [5]. This uses task duplication for weighted *DAG*s to be executed on an unbounded number of fully connected processors. As in DSH and BTDH, ancestors of tasks in the *critical path* are duplicated on idle slots if tasks on the *critical path* start earlier. Priorities are given to the tasks: a task is a *very-important-parent* of a task v if it is the immediate predecessor of v that sends the message last received by v. The *very-important-parent* task is replicated together with v. Then, recursively from the parent tasks the same duplication technique is applied whenever tasks on the *critical path* can start earlier.

3.3.2 An abstract communication overhead

The delay on the communication is considered by many researchers as the only important communication parameter of parallel architectures. When the *delay* is the only parameter considered, there is no need to know the number of *sending* or *receiving events* which are incurred when executing the application in parallel, since communication and computation can overlap. Papadimitriou and Ullman, in [105], considered the relevance of these *communication events*, which are incurred when communication is carried out, in conjunction with task replication. The *computational time* is the time necessary to compute a *DAG* when neglecting the communication delays for a given schedule, say, S. It is possible to derive a relation between the processors allocated to tasks taking into account their precedence relationship. As already specified in Section 2.3, the relation *D* represents this dependence between processors which is equivalent to the total amount of communication required to execute G considering S. Papadimitriou and Ullman relates the edges of *D* with the total amount of message traffic generated. Therefore, it can be used to model congestion rather than overheads. The edges are referred to events and thus, an *abstract* *event* model is defined. Also, Papadimitriou and Ullman define the *delay* as the number of edges on the longest path in D. This concept of delay is not the usual delay but it expresses the global delay which is incurred in the parallel execution of G. Adopting both characteristics defined in [105] as a scheduling model would require the implementation of scheduling strategies that have a "global" view of the application when the specification of the schedule is being established.

In the LPRAM model of Aggarwal, Chandra and Snir [4] (discussed in Section 2.2) the notion of locality is incorporated, since an access to the local memory associated with a processor has negligible cost while an access to the shared memory has a certain cost l. In a similar manner to the Papadimitriou and Ullman model, a processor in LPRAM can either be computing or accessing the global memory. The difference between the two models is that in LPRAM each processor can have at most one communication request outstanding at any time and task replication is not an issue. In LPRAM the communication is representing the number of events which are incurred when a processor communicates.

3.3.3 *CPU penalty* and delay

The overhead when sending or receiving messages is a characteristic that cannot be neglected. In multicomputers, the major drawback is to properly implement message passing mechanisms, due to the fact that the time that a processor is blocked using primitives like *send* and *receive* is very significant [63]. In the literature, there are scheduling models which incorporate the startup overhead in the communication cost. However, the majority of these models add this cost to the delay rather than considering it as a *CPU penalty*. We now discuss scheduling approaches that consider the startup overhead in their model as a *CPU Penalty*.

Lee and Hurson [81] proposed a method called the *Balanced Layered Scheme* BLS, which consists of three phases: *separation*, *assignment* and *ordering*. Firstly, in the *separation* phase, the *critical path* of G is identified and its tasks are said to belong to a single *virtual thread*. At each iteration, a *critical path* is identified and its tasks are enqueued and marked. At iteration i, a task v_j is dequeued and the longest directed path L_i emanating from v_j is specified; this path should only contain tasks not yet marked. As observed in [81], this method is similar to the linear clustering algorithm proposed by Kim and Browne [75]. After all the tasks have been marked, the *assignment* phase initiates where each virtual thread is assigned to a processor to produce the smallest communication cost. Finally, the *ordering* phase specifies the earliest start time of each task v_j (it is when the actual schedule time is defined), which depends on the finishing time of v_j's immediate predecessors and allocation. Whenever the task v_j and one of its immediate predecessor, say, v_k are allocated to distinct processors, the communication cost $C_{k,j}$ is considered as $C_{k,j} = t_s + ht_h + \frac{D_{k,j}}{BW}$, where, t_s is the startup overhead, t_h is the delay per hop, h is the number of hops between the communicating processors, $D_{k,j}$ the amount of data transfered and BW, the channel bandwidth. Lee and Hurson carried out experiments on the nCUBE 3, taking into account coarse grained applications. Compared with other strategies, BLS seems promising, but no strong evidence that it would work well with fine grained *DAG*s is shown. The possibility of combining various messages which are destined for the same processor are yet to be evaluated and designed. The examples showed tasks with low out degree and consequently, a low number of startup overheads is incurred.

In [42, 86], Lewis *et al.* discussed scheduling models which incorporate delay and startup overhead associated with the *sending* of messages. Initially, only the scheduling of *regular DAG*s are analysed, more specifically, the *replicate* graph, the *divide-and-conquer* and the *nearest-neighbour mesh*. The graphs are analytically evaluated, considering intuitive schedules and the number of processors which produces the minimal makespan is derived for given architectural parameters. Then, a *composite graph* is introduced as a task graph containing regular-structured subgraphs as components. Since the application of optimal schedules for each component does not guarantee a global minimum, but can lead to a local one, Lewis *et al.* propose to *flatten* the *composite* graph, that is, each task of each component is considered. At first, the best number of processors for the *composite* graph is defined as the summation of the optimal numbers of processors for each component (a regular structure) and then, the *composite* graph is flattened. Both results is given as input to a list scheduling algorithm, like Mapping Heuristic (MH) [40, 41]. However, MH considers that the startup overhead is added to the delay. Therefore, it is not clear if the startup overhead is considered as *CPU penalty*. Also, it is not always true that the evaluation on the number of processors proposed gives the best result. From the references provided by the authors, "further work is needed to extend the method to larger and more complex patterns of parallelism, and to calibrate parameters as process and message startup, processor speed, and transmission speeds of real machines. Also, further work is needed to study how to perform global optimisations" [86].

McCreary *et al.* [74, 97, 98, 99, 100], based on *graph decomposition* theory [122, 132], proposed a method which transforms a *DAG* onto a hierarchy (tree) of subgraphs, referred as *clans*, where the precedence relationship among tasks is represented. The top task of the tree represents the entire graph and its immediate successors represent *clan* subgraphs. The immediate successors of each *clan* in the tree are referred as its *subclans*. The necessary definitions used to construct *clans* are that all of the tasks within a *clan* must have the same communication requirements with tasks outside the *clan*. McCreary *et al.* performed a series of experiments comparing with other scheduling strategies that uses the *delay* model. Therefore, it is not clear if startup overhead is considered as a *CPU penalty*.

3.3.4 Discussion and remarks

In the light of the above classification and description of scheduling models and strategies, we see that the startup overhead is not usually considered as a *CPU penalty*. Apart from the heuristic proposed by Lee and Hurson [81, 82], which defines explicitly the startup overhead and mention, as future work, the possibility of "combining different messages destined for the same processor", the other scheduling models treat *CPU penalty* as part of the delay.

Even though Sarkar and Hennessy in [117] agree that "packing introduces a non-linearity in the communication costs", their proposed scheduling model ignores the CPU penalty caused by the sending of messages. Thus, the communication cost being represented by the linear function $\alpha + w\beta$, as already mentioned in above paragraphs, may not represent the reality.

One point to be remarked is related to the evaluation of the heuristics. We find in the literature experimental evaluations based on the comparison of schedules produced by proposed heuristics with schedules produced by already known heuristics. However, it is also important to know how the respective schedules would perform on a realistic machine.

We find in the literature criticisms on the utilisation of task replication. For example, McCreary *et al.* [74] does not allow duplication of tasks on different processors, when defining their *graph decomposition method*. Their argument is that "duplication adds additional complexity on an already intractable problem that none of our competing methods use". It seems that their main worry is the possible high complexity of a proposed heuristic which adopting replication of tasks can have. Discussing the same subject, Gerasoulis and Yang *et al.* do not allow duplication of tasks, since "it results in an increase in the space complexity since data must also be duplicated" [47]. However, depending on the communication costs, replication can provide great benefits [103], as for example, decreasing the number of times that communication events are incurred when executing an application in parallel. This number of *communication events* can reflect a huge charge to the processors, in the form of CPU penalty. It may be the case that without replication, the fastest way of executing the application is sequentially, and therefore, all the data associated with the tasks would be accessed by the same processor.

Complexity-reasonable algorithms based on task duplication can be found in the literature. Papadimitriou and Yannakakis [106] proposed a polynomial time algorithm which replicates tasks in order to minimise the communication costs. Darbha and Agrawal in [36] proposed a duplication strategy for *DAG*s, with variable execution time tasks, with complexity $\mathcal{O}(n+e)$, where n is the number of tasks and e is the number of edges of the task graph. Also, Ahmad and Kwok in [5] based their work on the duplication of tasks, since it can be very useful for systems with high communication overhead such as network of workstations. The DSH algorithm of Kruatrachue and Lewis [78] allows duplication of tasks whenever in order to produce a schedule with smaller makespan and its complexity is $\mathcal{O}(n^4)$. With the same complexity, an extension of DSH was proposed by Chung and Ranka [25].

In summary, considering the CLAUD model as our scheduling model, we need to define a scheme that can produced a "good" schedule of a given application on a parallel computer.

3.4 Application dependent heuristics

In the literature, we can find proposed heuristics together with their scope of efficiency, and in many cases, it is shown that this efficiency is closely related to characteristics of the application. One of the characteristics that determine the relation between the application and the target architecture is *granularity* of the application.

The granularity of a *DAG* $G=(V,E)$ is defined as the relation between computation costs and communication among tasks of G when executing it on the target architecture. Therefore, it is important to define the tasks execution costs and precedence relationship, the amount of data exchanged between adjacent tasks, and the significant architectural parameters. A *DAG* is said *coarse-grained* if computation dominates communication, otherwise it is called *fine-grained*. Stone has already asserted in [121] that the ratio between the computation and communication establishes whether or not it is advantageous to parallelise the application. Categorising the *DAG* as coarse-grained or fine-grained may correspond to the fact that the application is being executed by a small or large number of processors, respectively.

Not only the granularity of applications can be established, but also, machines are classified as coarse or fine grained. A parallel computer in which the processors sustain a high degree of computation efficiently are called coarse-grained. On the other hand, fine-grained machines are the ones that perform better for applications that requires a large number of processors, and the best machine performance is achieved when a high number of processors are involved in the computation.

The efficiency of many scheduling algorithms are evaluated in accordance to the granularity of the applications, as seen in some results summarised from the literature and discussed in Section 3.3. Therefore, granularity is also defined here. However, the usual definition of granularity may not express the exact relation between communication and computation costs for an application. Thus, a more strict definition of granularity is proposed. Having established this characteristic of the input application, a list scheduling framework is proposed and analysed, so that the intrinsic relationship between application and its structure, architecture parameters and algorithm is examined. Finally, a general list scheduling heuristic is evaluated and its applicability to the CLAUD model is established. The heuristic analysed is the *Earliest Task First* (ETF) schedule proposed by Hwang *et al.* [65]. The ETF algorithm not only considers communication costs, but also, the representation of the topology of the target architecture. This representation can be also be interpreted as a way to express the different amount of data exchanged between tasks of an application. These issues are analysed and important considerations established.

3.5 On the Definition of Granularity

The relation between each task $v_i \in V$ and one of its successors $v_j \in V$, both of them allocated to distinct processors is analysed. Let $odg(v_i)$ and $idg(v_j)$ be the out degree of task v_i and in degree of v_j, respectively. The total latency time required for v_i and v_j to communicate is

$$C_{(v_i,v_j)} = w((v_i, v_j)) \cdot \varphi(proc(v_i), proc(v_j))$$

the communication cost depending on which processors the tasks are allocated to. An upper bound on the communication between any two adjacent tasks is established in the following Lemma.

Lemma 3.5.1 *Let $G = (V, E)$ be a DAG scheduled under the* CLAUD *model. The upper bound on the communication time $CMax_{(v_i,v_j)}$ between two tasks $v_i, v_j \in V$, such that $(v_i, v_j) \in E$ is*

$$CMax_{(v_i,v_j)} \leq C_{(v_i,v_j)} + odg(v_i)\lambda_s + idg(v_j)\lambda_r$$

Proof:
In the worst case suppose that the message (v_i, v_j) is the last one to be sent from v_i considering all the $odg(v_i)$ messages. Also, suppose that this message is the last one to be received by v_j between all the $idg(v_j)$ input messages for v_j. Thus, before v_i sends its data to v_j, $odg(v_i)$ overheads are incurred in the processor in which v_i is allocated. Also, in the worst case $idg(v_j)$ *receiving events* are incurred on the processor allocated to v_j, due to messages received from immediate predecessors of v_j allocated to distinct processors.

□

The Lemma above states the upper bound on the elapsed time from when v_i finishes its computation until an immediate successor v_j starts its computation. Having defined the maximum communication time that can be incurred between two adjacent tasks, the granularity of a *DAG* is defined considering the architectural parameters of the target model [47, 131].

Definition 3.5.1 *Consider $m(v_i)$ the execution time of a task v_i of a DAG $G = (V, E)$, $pred(v_i)$ the set of immediate predecessors of v_i and $CMax_{(v_l,v_i)}$ the maximum communication time between $v_l \in pred(v_i)$ and v_i, including overheads and latency due to communication. The granularity of v_i is defined as*

$$g(v_i) = \frac{\min_{v_l \in pred(v_i)}\{m(v_l)\}}{\max_{v_l \in pred(v_i)}\{CMax_{(v_l,v_i)}\}}$$

The granularity of the DAG is then

$$g(G) = \min_{v_i \in V}\{g(v_i)\}$$

A DAG $G = (V, E)$ with $g(G) > 1$ is considered a coarse grained DAG, otherwise it is a fine grained.

This definition, although the usual one, may not reflect the exact characteristics of the given *DAG*. For example, consider the *DAG* G in Figure 3.2 with all tasks allocated to distinct processors which are fully connected. The latency associated with the links is unit and the overhead parameters are negligible ($\lambda_s = \lambda_r = 0$). In this case, $g(G) = 2/5$ and thus, the *DAG* is considered as a fine grained one. However, there exits a high execution cost associate with the majority of the tasks. Different algorithms have their behaviour analysed

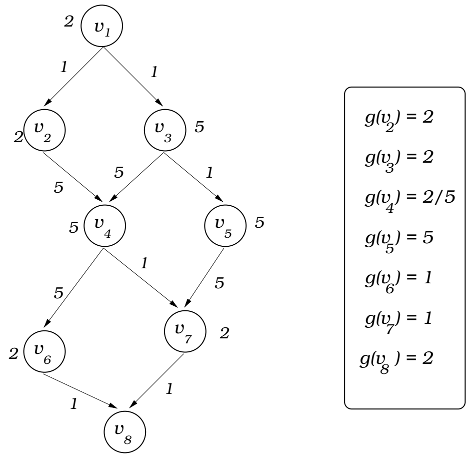


Figure 3.2: The *granularity* of each task in the *DAG* $G = (V, E)$ given the communication parameters $\tau = 1$, $\lambda_s = \lambda_r = 0$. The target network is fully connected.

in accordance to different kinds of *DAG*s which are classified using their granularity properties. However, following the definition above, if the algorithm used is proved to have a good behaviour on a coarse-grained *DAG*, the schedule produced for the Example *DAG* may be a unexpected one and far from the optimal.

A more strict classification of granularity of *DAG*s is proposed below.

Definition 3.5.2 *The granularity of a vertex $v_i \in V$ is specified as in Definition 3.5.1:*

$$g(v_i) = \frac{\min_{u \in pred(v_i)}\{m(u)\}}{\max_{u \in pred(v_i)}\{CMax_{(u,v_i)}\}}$$

A DAG $G = (V, E)$ is considered complete coarse-grained if $g(v_i) > 1, \ \forall v_i \in V$. If $g(v_i) < 1, \ \forall v_i \in V$, the DAG is complete fine-grained. If both characteristics do not apply to all tasks in G, the graph is then classified as coarse or fine-grained in accordance with Definition 3.5.1.

3.5.1 Grain Dependent Strategy

A framework for the scheduling strategy for the CLAUD model is specified, considering the list scheduling strategy in which the priority is dictated by the earliest start time, the amount of data exchange between tasks and a task's immediate predecessors. Given as input a *DAG*, $G = (V, E)$, and the communication parameters of the target architecture, the *list scheduling framework* produces a schedule S, in which the *communication event* tasks are also scheduled. The framework and important definitions are described as follows and it is implemented in Algorithm 2. We define, since there is no replication of tasks, $s(v_i)$ as the start time of a task $v_i \in V$ and $f(v_i)$, its finish time. The processor to which v_i is allocated is given by $proc(v_i)$, considering that there are as many processors as the number of source tasks. For each task v_i scheduled and for each communication event incurred, a tuple is added to S. Note that $first(l)$ is a procedure that removes the first element of the list l and returns this element.

Initially, $S = \emptyset$, all processors are considered *idle* and the source tasks are *free* to be scheduled. Each source task is allocated to a distinct processor and scheduled at time step 0. At each iteration of Algorithm 2, a list of *free* tasks is available in A and ready to be scheduled. Each task $v \in A$ is scheduled on the same processor as the immediate predecessor u' from which v receives the respective data at the latest time. This immediate predecessor u' is associated either with a large execution cost or high communication cost, if compared with the other immediate predecessors of v. In order to identify u', the immediate predecessors of v are given in the decreasing order of execution and communication costs in the list $plist(v)$. The next immediate predecessor in $plist(v)$, say u, is determined, in order to calculate the start time of v, since u is the immediate predecessor in $plist(v) - \{u'\}$ which causes v to start the latest. The time of v is initially predicted but it is not definitive, because the *receiving events* overheads are yet to be specified. A *sending event* task is scheduled due to (u, v) if u and v are allocated to distinct processors. After incorporating the tuple associated with $\sigma_{(u,v,proc(u))}$ to S, λ_s is incorporated in the schedule with the finish time $f(u)$. This allows the proper schedule of other *sending event* tasks which are incurred when the task u sends messages to immediate successors other than v, which are allocated to processors different from $proc(u)$. Depending on their values, these sending overheads can be overlapped with the execution of the immediate predecessor u'. In order to more accurately schedule the *receiving events* tasks on $proc(v)$, for each $\rho_{(u,v,proc(v))}$, $u \in pred(v)$, the overhead λ_r is added to the start time.

Algorithm: 2 *List scheduling framework* {

$q := 0;\ \ A' := \emptyset;\ \ S := \emptyset;\ \forall v \in V,\ D(v)\ := \mid pred(v) \mid;$
$I := \{p_0, p_1, \ldots, p_{|\mathcal{P}|}\};\ \ A := \{v_i\ :\ pred(v_i) == \emptyset\};$
$\forall v_i \in A,\ p_j \in I$ {
  $proc(v_i) := p_j;\ \ s(v_i) := 0;\ \ f(v_i) := m(v_i);$
  $S := S \cup (v_i, p_j, 0);$
  $A := A - v_i;\ \ I := I - p_j;\ \ q := q + 1;$
  $\forall u \in succ(v_i),\ D(u)\ :=\ D(u) - 1;$
  $A' := A'\ \cup\ \{u \in succ(v_i)\ :\ D(u) = 0\};$
}
while $(q\ \leq \mid V \mid)$ {
  $A := A';\ \ A' := \emptyset;$
  while $(A \neq \emptyset)$ {
    $v := first(A);$
    $plist(v) :=\ \ \{u_i\ :\ u_i \in pred(v)\ \ \wedge\ \ f(u_i) + C(u_i, v) \geq f(u_{i+1}) + C(u_{i+1}, v),\ i = 1, \ldots, \mid pred(v) \mid\};$
    $u' := first(plist(v));\ \ u := first(plist(v));$
    $earliest := \max\{f(u'), f(u) + C(u, v)\};$
    $\forall u \in plist(v)\ :\ proc(u) \neq proc(v)$ {
      $S := S \cup (\sigma_{(u,v,proc(u))}, proc(u), f(u));$
      $f(u) := f(u) + \lambda_s\ ;$
      $earliest := \max\ \{earliest, f(u) + C(u, v)\}$
      $earliest := earliest + \lambda_r;$
      $S := S \cup (\rho_{(u,v,proc(v))}, proc(u'), earliest);$
    }
    $S := S \cup (v, proc(u'), earliest);$
    $proc(v) := proc(u');\ \ s(v_i) := earliest;\ \ f(v_i) := s(v_i) + m(v_i);$
    $q := q + 1;$
    $\forall u \in succ(v_i),\ D(u)\ :=\ D(u) - 1;$
    $A' := A'\ \cup\ \{u \in succ(v_i)\ :\ D(u) = 0\};$
  }
}
}

Having specified the algorithm which considers the communication parameters defined in the CLAUD model, we now establish its optimality, which, as we will see, depends on the application structure and its granularity. It is interesting to analyse the difficulties that may arise when dealing with scheduling models containing communication parameters that are distinct in their nature. In the following Theorem, considering the input application to be a complete coarse grained Tree, we see that the *list scheduling framework* produces the optimal schedule, if the number of physical processors is no less than the width $\mathcal{W}(G)$ of the *DAG* G (the number of source tasks).

Theorem 3.1 *Algorithm 2 determines an optimal schedule for a complete coarse-grained Tree* $G = (V, E)$*, on a fully connected network with* $m \geq \mathcal{W}(G)$ *processors, considering the* CLAUD *model.*

Proof:
Algorithm 2 specifies that each source task is allocated to a distinct processor and each non-source task $v \in A$ is scheduled together with the predecessor u' with the highest value $f(u') + C(u', v)$ in $plist(v)$. Considering complete coarse-grained Trees, there is no gain to schedule on the same processor $proc(v)$ the next immediate predecessor u_1 in the list $plist(v) = \{u', u_1, \ldots, u_{|pred(v)|-1}\}$. Supposed that there is a gain if the next immediate predecessors $u_1 \in plist(v)$ is allocated to the same processor $proc(u')$, i.e. this allocation allows the task v to start earlier if u_1 is allocated to the same processor as u'. Therefore, $f(u') + m(u_1) < f(u_1) + C(u_1, v)$, that is

$$f(u') - C(u_1, v) < s(u_1) \tag{3.1}$$

The ordered list $plist(v)$ in line 13 establishes that $f(u') + C(u', v) \geq f(u_1) + C(u_1, v) \geq \ldots$ and therefore,

$$f(u') - C(u_1, v) \geq f(u_1) - C(u', v) \tag{3.2}$$

Considering a complete coarse-grained *DAG*, $\forall v \in V,\ g(v) > 1$, which is equal to

$$\min_{u \in pred(v)} \{m(u)\} > \max_{u \in pred(v)} \{CMax(u, v)\} \tag{3.3}$$

It is straight forward to derive that

$$m(u_1) \geq \min_{u \in pred(v)} \{m(u)\} \quad \text{and} \quad C(u', v) \leq \max_{u \in pred(v)} \{CMax(u, v)\}$$

Substituting in 3.3, it follows that $m(u_1) - C(u', v) > 0$ which is equivalent to

$$s(u_1) + m(u_1) - C(u', v) > s(u_1) \tag{3.4}$$

Therefore, from inequalities 3.2 and 3.4, we have $f(u') - C(u_1, v) > s(u_1)$ which contradicts the inequality 3.1.

□

3.6 A Non-uniform Delay Based Heuristic

Hwang *et al.*, in [65], specified a heuristic called ETF (*Earliest Task First*) to schedule a set of tasks (with an arbitrary precedence relationship between them) onto a set of processors taking into account non-uniform communication delays. The heuristic is considered a realistic in the sense that, it not only incorporates communication costs, but also, some notion of the topology. A function that depends on the processors to which the communicating tasks have been allocated, represents the communication delay between processors. Also it is possible to represent different messages' lengths, transmitted in the processor network by this function.

A *DAG* $G = (V, E)$ is used to represent the application to be scheduled, with each tasks having an associated execution cost that is not necessarily unit, denoted by a function $m : V \to Z_0^+$ that is used to assign execution cost values $m(v_i)$ to each vertex $v_i \in V$. Furthermore, the amount of data to be exchanged between adjacent tasks is expressed by the function $w : E \to Z_0^+$ which assigns a weight value $w((v_i, v_j))$ to the edge $(v_i, v_j) \in E$. The network topology is modelled by the function $\varphi : \mathcal{P} \times \mathcal{P} \to Z_0^+$, which represents the variation in communication delays between different pairs of processors, as already defined in Section 3.1.1.

Hwang *et al.* analysed the ETF algorithm and concluded that the makespan $\mathcal{M}(S_{ETF})$ of the produced schedule S_{ETF} is bounded on the Graham's bound for the list scheduling and communication requirements [54], i.e.

$$\mathcal{M}(S_{ETF}) \leq (2 - \frac{1}{n})T_{opt} + \textit{Communication Requirements} \tag{3.5}$$

where T_{opt} is the makespan of the optimal scheduling ignoring the interprocessor communication. The first term of the right side of Equation 3.5 is the Graham's bound. The *Communication Requirements* term is not supposed to have a impact on the performance of S_{ETF}.

Next, an evaluation of the list scheduling algorithm ETF, which considers the amount of data exchanged between tasks is carried out. The main objective of this analysis is to test the applicability of this kind of list scheduling algorithm in more realistic models.

3.6.1 The ETF algorithm

When allocating vertices to processors, the ETF algorithm takes into account the amount of data exchanged between different tasks. A vertex v_i is *free* to be allocated to an *idle* processor once all of its predecessors have been assigned to some processor. The ETF algorithm checks all the idle processors at the *current moment* and chooses the processor on which the *free* vertex v_i starts earliest, on the attempt to minimise the resulting makespan. ETF does not deal with the replication of tasks and when adjacent tasks are allocated to the same processor, the communication cost is zero. Whether two tasks should be allocated to the same or distinct processors is determined by the algorithm which relies on the overlapping of computation and communication as a strategy to minimise the makespan of the resulting schedule.

3.6.1.1 Describing ETF in more details

The ETF algorithm is shown in Algorithm 3. From line 1 to 3 the main control variables are initialised. The variable q controls the number of tasks which have been schedule. The $D(v_i)$ keeps the number of predecessors of the task v_i which have been scheduled. When $D(v_i)$ is zero, v_i is *free* or *available* to be scheduled. The set of tasks A contains those tasks which are *available* to be scheduled. Each *available* task is allocated to idle processors. The set of idle processors at the current moment is denoted by I. The variable CM, the *current moment*, is the time that busy processors "finish" to execute their just allocated task and are able to be included in the set I, i.e. CM represents the time that processors become *idle*. It is also necessary to keep the earliest time after CM that currently busy processors will become *idle*. This time is called the *next moment* and is denoted by NM.

The first tasks to be scheduled are the sources which are scheduled at time step zero. The start time of a task v_i is denoted by $s(v_i)$. After the sources are scheduled, their immediate successors are *free* to be scheduled and are included in the set A. When either A or I is empty and there remain tasks to be scheduled, the lines 13 to line 16 performs the search for processors which are idle at time step NM. In order to schedule a *free* task $v_i \in A$ on an *idle* processor $p_j \in I$, the earliest time that v_i can be executed on p_j, is calculated in line 24 and represented by the cost function:

$$r(v_i, p_j) = \max_{v_l \in pred(v_i)} (f(v_l) + w((v_l, v_i)) \cdot \varphi(proc(v_l), p_j))$$

where $f(v_l)$ is the time when the immediate predecessor v_l of v_i finishes execution and $w((v_l, v_i))$ is the data volume transmitted through the edge (v_l, v_i).

Algorithm: 3 *ETF* ($G = (V, E), \varphi : \mathcal{P} \longrightarrow Z_0^+$) {

1 $\quad CM := 0;\ NM := \infty;\ q := 0;\ NoStop := 1;\ D(v_i) := \mid pred(v_i) \mid, \forall v_i \in V;$

2 $\quad I := \{p_0, p_1, \ldots, p_m\};\ A := \{v_i \mid D(v_i) == 0\};\ f(v_i) := \infty, \forall v_i \in A;$

3 $\quad \forall v_i \in A, p_j \in I\ \ r(v_i, p_j) := 0;$

4 $\quad$ while ($q < \mid V \mid$){

```
        while ((I ≠ ∅) AND (A ≠ ∅) AND (NoStop)){
            v', p' : min_{v_i ∈ A} min_{p_j ∈ I} r(v_i, p_j);
            early := max(CM, r(v', p'));
            if (early ≤ NM){
                proc(v') := p'; s(v') := early; f(v') := s(v') + m(v');
                A := A − v'; I := I − p';
            }else {
                NoStop := 0;
            }
        }
        CM := NM; F := {v_i | f(v_i) == CM, v_i ∈ V};
        NM := next(CM); A' := ∅;
        for v_i ∈ F{
            I := I ∪ p(v_i);
            for v_j ∈ pred(v_i){
                D(v_j) := D(v_j) − 1;
                if (D(v_j) == 0)
                    A' := A' ∪ v_j;
            }
        }
        A := A ∪ A';
        for v_i ∈ A{
            for p_j ∈ I{
                r(v_i, p_j) := max_{v_j ∈ pred(v_i)}(f(v_j) + w((v_j, v_i)) * φ(p(v_j), p_j));
            }
        }
    }
}
```

3.6.2 ETF partition on the *diamond DAG*

The $n \times n$ diamond *DAG* represents an important class of scientific computation, namely computational analysis, and is representative of applications such as those used for computing the longest common subsequence or a subcomputation in LU decomposition [102, 104], systolic arrays and matrix multiplications [4, 61, 90, 105].

A $n \times n$ diamond *DAG* is a $n \times n$ grid, in which the vertices are labeled as the Cartesian coordinates, each row being the x axis and each column the y axis, as shown in Figure 3.3. The lines are labeled, starting from the bottom to the top of the grid, from o to $n-1$ and the columns, from the left to the right side, from 0 to $n-1$.

Each vertex $(i, j),\ i = 0, \ldots, n-2,\ j = 0, \ldots, n-2$, have two immediate successors $(i, j+1)$ and $(i+1, j)$. Each vertex $(n-1, j)$ has only one immediate successor $(n-1, j+1),\ j = 0, \ldots, n-2$. Each vertex $(i, n-1)$ has $(i+1, n-1),\ i = 0, \ldots, n-2$ as the only one immediate successor.

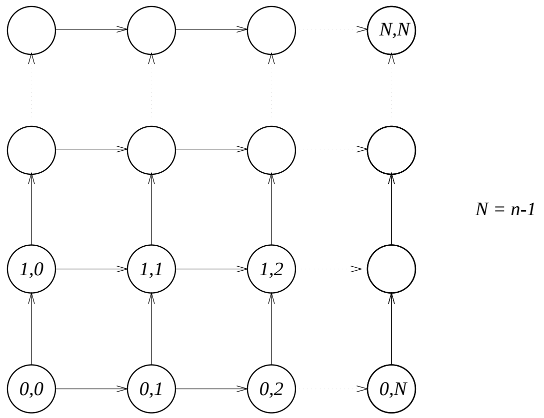


Figure 3.3: The $n \times n$ *diamond DAG*

The application and architecture characteristics which are considered are described as follows. The execution cost associated with each vertex is unit, so that $m(v_i) = 1,\ \forall v_i \in V$ and the amount of data associated with each edge is equal to a certain value, say, δ, i.e. $w((v_i, v_j)) = \delta,\ \forall (v_i, v_j) \in E$.

The target machine is supposed to be a fully connected interconnection network with m processors listed in the set $\mathcal{P} = \{p_0, p_1, \ldots, p_m\}$. The latency associated with the communication between any two adjacent processors is unit, i.e. $\varphi(p_i, p_j) = \tau = 1,\quad p_i, p_j \in \mathcal{P},\ p_i \neq p_j$. The reason for these assumed values is due to the objective of evaluating the performance of this list scheduling algorithm on a *DAG* when long messages are in transit. In this way, it is possible to establish the usability of such algorithm when scheduling applications considering models, like the CLAUD model, which represents a wide range of parallel machines, include the ones with high granularity.

3.6.2.1 Executing the algorithm

The first task scheduled, the source (0,0) is allocated to any processor in the initial set of idle processors, the set $I = \{p_0, \ldots, p_m\}$, and thus, any processor can be selected. Suppose (0,0) is scheduled on the processor p_0 at time $s((0, 0)) = 0$. As the set of *free* tasks A becomes empty, the search of new *free* tasks is carried out by executing lines 15 to 24, in which it is verified that tasks (0,1) and (1,0) become *free*, since their immediate predecessor $(0, 0)$ has just been computed. Therefore, $A = \{(0, 1), (1, 0)\}$ and the processor p_0 becomes idle again.

Although both tasks, if scheduled on processor 0, are executed at time 1 (as specified in the cost function r in line 24), it is obvious that just one can be selected. The selection is carried out in such a way that all tasks of the same row are scheduled on the same processor. This priority does not change the ETF algorithm, as the breaking of ties is arbitrary when selecting *free* tasks. Therefore, the task first executed is the (0,1). At this current moment, the task (1,0) could be scheduled at time $1 + \delta$ on any other processor. However, the variable NM indicates that the (1,0) can be executed at time step 2 on processor p_0 (same processor as its immediate predecessor), as seen in line 8. Therefore, if $\delta > 1$, no action is taken at this current moment CM, and the sets A and I are updated again (performing lines 15 to 24). On the other hand, note that task (1,0) would be scheduled at this point if $\delta = 1$. After (0,1) is computed, tasks (0,2) and (1,1) become free and processor 0 becomes idle, so that $A = \{(1,0),(0,2),(1,1)\}$ and again, all of the processor in $\mathcal{P}$ are idle. All tasks $v_i \in A$ can be scheduled at time step 2 on processor p_0, the one on the same line as $(0,1)$, the task (0,2), is the next to be started on processor p_0.

Suppose that at time step t the set of tasks which are scheduled are those (i,j) such that $t = i\cdot(1+\delta)+j$, for $i = 0,\ldots,n-1$. At time $t+1$ all busy processors become idle again (because the task execution cost is unit) and considering that each task (i,j) just computed is not the last task of the respective row i, has as immediate successors the task $(i,j+1)$ and $(i+1,j)$, which become *free*. The tasks $(i,j+1)$, on the same row as (i,j), can either be scheduled on the same processor $proc((i,j))$ at time step $t+1$ or on any other processor at time step $t+1+\delta$. Therefore, $(i,j+1)$ is scheduled on $proc((i,j))$ at time $t+1$, as it is the earliest start time in accordance to the cost function $r((i,j+1),p),\ \forall p \in I$ (line 6). Consequently, at time step $(i+1)\cdot(i+\delta)$, the task $(i+1,j)$ is scheduled any other (and only one) processor. Therefore, at every $(1+\delta)$ time steps the task $(i+1,j)$ is schedule on a distinct processor from its immediate predecessor (i,j) allocated one. After the last task of the row i is computed, the task $(i,n-1)$, its processor $proc((i,n-1))$ returns to the set of idle processors I and there will not be any other tasks on the same row to be scheduled on this processor. Therefore, at most $m_{max} = \left\lceil \frac{n}{1+\delta} \right\rceil$ processors are involved in the overall computation, i.e. there is no improvenment on the makespan if more than m_{max} processors are used.

The above discussion of the ETF algorithm behavour is established in the following Theorem, which has its formal proof in [18] equivalent to the above explanation:

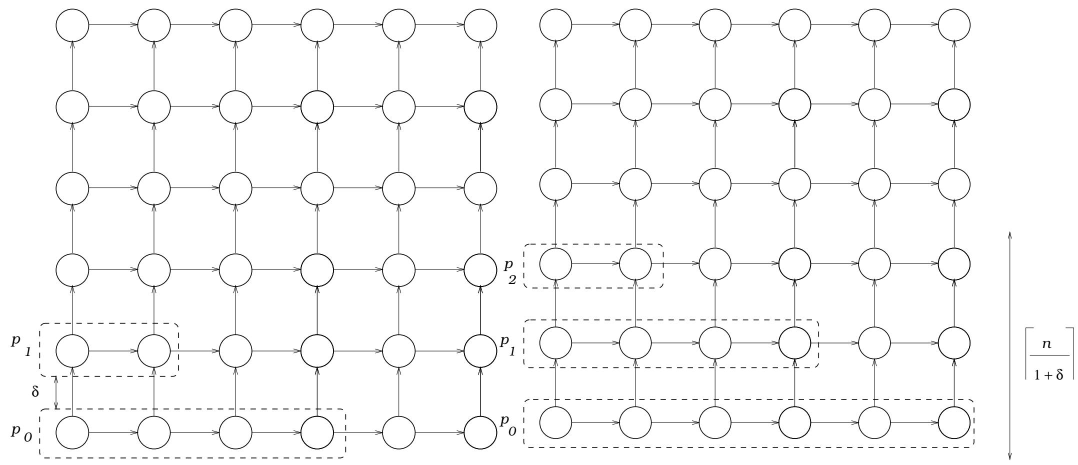


Figure 3.4: ETF partitions the $n \times n$ *diamond DAG* into lines

Theorem 3.2 *The ETF algorithm schedules all task on the same row of the diamond DAG on the same processor, so that the task (i, j) is scheduled at time $i \cdot (1 + \delta) + j$. The number of processors that may be involved in the computation simultaneously is $\left\lceil \frac{n}{1+\delta} \right\rceil$.*

From Theorem 3.2, the makespan $\mathcal{M}(S_{ETF})$ of the strategy describe above is given by the finishing time of the sink $(n-1, n-1)$, considering the maximum number of processors $m_{max} = \left\lceil \frac{n}{1+\delta} \right\rceil$, i.e. :

$$\mathcal{M}(S_{ETF}) = n + (n-1) \cdot (1 + \delta)$$

Norman *et al.* [104] observes some interesting properties when partition the diamond *DAG* by lines and stripes. From their evaluation of the line and stripe partitioning strategy on the diamond *DAG*, the number of communication events when partitioning by lines is much greater than when partitioning by stripe. Therefore, in a system with a high startup overhead, the line partition leads to a schedule with higher makespan.

An interesting point to be observed is that, considering the makespan for the line partition evaluated in [104] to be

$$\mathcal{M}(S_{lines}) = \frac{n^2}{m} + (m-1)(1 + \delta)$$

the minimum is obtained when the number of processors is

$$m_{min} = \frac{n}{\sqrt{1 + \delta}}$$

However, when applying the ETF algorithm there is no improvement on the resulting makespan $\mathcal{M}(S_{ETF})$ if more than $\left\lceil \frac{n}{1+\delta} \right\rceil$ are used. Therefore, the possible minimum for the line partition is not achieved when using the ETF algorithm.

Two important facts are inferred from the above evaluation. Firstly, an interval of optimality for the ETF schedule can be identified, which is dictated by the δ value, i.e. whenever $0 \leq \delta < 1$. If there is no delay on the communication the processors can start to execute its row as soon as the first task of each row is computed which is the optimal behaviour, so that the resulting makespan is equal to $2n$, the optimal makespan. Secondly, when δ is too large, say, $\delta > n-1$. only one processor is used in the schedule and the makespan is n^2.

3.6.3 ETF and the Binary Tree

The Full Binary Tree $B_n = (V_B, E_B)$ is a *DAG* with n tasks, in which the direction of the edges in E_B are towards the unique sink in V_B. Each vertex has one immediate successor (apart from the sink vertex) and two immediate predecessors (apart from the source vertices). In a binary tree B_n, there exists is $\frac{n+1}{2}$ source tasks; the height of the tree is $l = \log(n+1)$; the tree is divided in levels, where the sources are at level $l-1$ and the sink at level 0; and each level contains 2^i tasks, $0 \leq i < l$.

Suppose that the tasks in B_n have associated a unit execution cost and that the amount of data transmitted through each edge $(v_i, v_j) \in E_B$ is equal to a certain value δ. i.e $m(v_i) = 1,\ \forall v_i \in V_B$ and $w((v_i, v_j)) = \delta,\ \forall (v_i, v_j) \in E_B$. The ETF algorithm is executed having as input the above specified Binary Tree B_n, considering a fully connected network with m processors in the set $\mathcal{P} = \{p_0, p_1, \ldots, p_m\}$, with latency on the communication equal to $\varphi(p_i, p_j) = \tau = 1,\ p_i, p_j \in \mathcal{P},\ p_i \neq p_j$.

3.6.3.1 Schedule produced for B_n by ETF

The $\frac{n+1}{2}$ source tasks are initially in the set of *free* tasks A and all the m processors are in the set I. Using more than $\frac{n+1}{2}$ processors, will not provide a better makespan, since any of the source tasks in A can be selected and scheduled on any of the processors in I at time step 0, as initially, the cost function $r(v_i, p_j) = 0,\ \forall v_i \in A,\ \forall p_j \in I$. As the ETF algorithm arbitrarily schedules tasks when there are ties, considers the situation in which two tasks v_i and v_l are allocated to two distinct processors p_j and p_k, respectively. Suppose that these two tasks are such that they have the same immediate successor v_w. After both v_i and v_l, their processors are inserted in the set I since they become idle (line 16) and the immediate successors v_w becomes a *free* task (line 21). When calculating the cost function, it is concluded that $r(v_w, p) = 1 + \delta$, for any $p \in I$. This critical situation can be propagated until the sink task is scheduled, the resulting in a makespan is equal to

$$\mathcal{M}(S_{ETF}) = \log(n+1) + (\log(n+1) - 1)\delta$$

As we will see in Chapter 9, the partition of the Binary Tree provides means to define an optimal makespan, considering that the delay is the only significant parameter. Also, Papadimitriou and Yannakakis, in [106], proposed an asymptotically optimal schedule for the Binary Tree, based on the replication heuristic, a result which was testified by Jung *et al.* in [70]. Comparing the two strategies, the ETF approach can only give good results when δ assumes a negligible value.

3.7 Conclusions on the applicability of ETF

In the light of the CLAUD model, if the ETF algorithm is used as a platform for scheduling an application, the result produced can be far from the optimum depending on the granularity of the application and of the target architecture. If long messages are sent and the application is fine grained, inefficient results are achieved such as the one described above. Also, the ETF algorithm is based on a model that does not take into account the overhead parameters λ_s and λ_r defined in the CLAUD model, parameters which cannot be neglected when considering a wide range of parallel machines.

ETF is a fair representative of list scheduling algorithms. Optimal decisions are made locally rather than globally and consequently, critical results may be produced when the communication costs have significant value. In [9], assuming that the number of available processors is arbitrarily large and communication delays are shorter than task execution times, it was shown that the makespan of the schedule produced by ETF is no more than a factor of three of the optimal makespan. Taking into account these results, in order to take advantages of the benefits the ETF can provide, a fine grained application should be transformed in such way that a coarser application is produced.

Still, the problem of scheduling a given application onto a network of processors with arbitrary topology is yet to be explored. Although ETF was proposed to reasonably schedule *DAG*s considering topologies issues, with the above evaluations, the applicability of such strategy is not very convincing when topological aspects are very important. What is more, even if the network is fully connected, long messages exchanged among tasks are not likely to be handled efficiently.

Chapter 4

A Multi-stage Approach: A new strategy

Considering the CLAUD model and parallel machines in which high overheads are incurred on processors when sending or receiving messages, we need a scheduling strategy that deals with these type of communication parameters in a reasonably efficient manner. On the evaluation of list scheduling algorithms like ETF, we conclude that this class of scheduling heuristics are not capable to provide a good schedule taking into account fine grained applications and a CLAUD "machine".

Also, when devising a strategy to schedule a given application, it might not be straightforward to consider all of the architectural parameters at once, due to the differences in the parameters nature. For example, while the overhead parameter is a CPU activity which disables the processor from executing any other task, the delay on the communication is a parameter that does not "block" the processor in such a way. One of the reasons for the difficulty of devising procedures which consider all the parameters at once is the computational complexity.

A *multi-stage approach* is proposed where, at each stage, a subset of the list of parameters characterising the architectural model is taken into account, resulting in a series of transformations performed on the input *DAG* $G = (V, E)$ until its schedule on the target machine is well established. The *multi-stage approach* aims to produce a good schedule for general applications considering target architectures where the communication overheads are critically high.

Below, Figure 4.1 shows a diagram of the whole strategy, its inputs and implemented heuristics. Next section, we present the new approach, defining some important concepts and outlining the main mechanisms. The details of each stage will be seen in the next chapters of this dissertation.

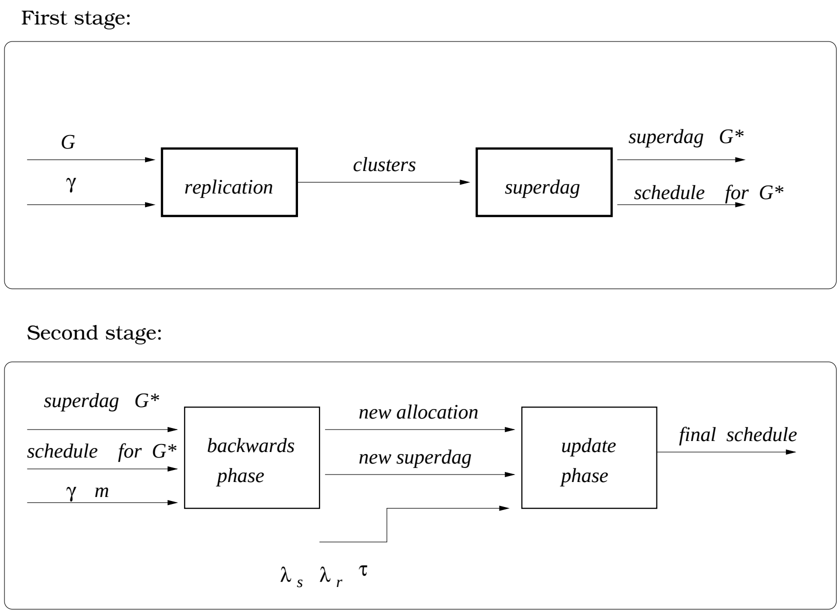


Figure 4.1: First stage: a two-algorithms stage. Second stage: the final schedule.

4.1 Communication events: a minimisation

The first parameters to be considered are the overheads λ_s and λ_r associated with the *sending* and *receiving* events, respectively, events which corresponds to CPU activity on the *sending* and *receiving* processors. When sending (or receiving) a message, a processor $p \in \mathcal{P}$ takes λ_s time units to prepare the packing (or unpacking) of a message, either small or large (although, some variable cost can be added to the overhead when very large messages are sent). We refer to the amount of this CPU activity as a *blocking parameter*, due to the fact that the processor is not able to execute any other kind of tasks while the respective event is being carried out.

In order to know the impact that the *communication events* may have on the performance of S, let us analyse a given schedule S of a small example *DAG* G. Having in mind the characteristics of the CLAUD model, a better schedule of the same *DAG* G might be provided. Let $G' = (V', E')$ be a subgraph of G shown in the Figure 4.2 (in fact, this is a critical allocation, but can be produced by a given heuristic which takes decisions up to the local stage of the current scheduling process), in which $V' = \{v_i, v_l, v_j\}$ and $E' = \{(v_i, v_j), (v_l, v_j)\}$. Suppose that, in the given schedule S, the two adjacent tasks $v_i, v_j \in V$ are allocated to distinct processors p and p' such that $(v_i, p, s(v_i, p)), (v_j, p', s(v_j, p')) \in S$: where $p \neq p' \in \mathcal{P}$ and

$$s(v_j, p') \geq s(v_i, p) + m(v_i) + \lambda_s + \lambda_r + w((v_i, v_j)) * \varphi(p, p')$$

Also in S, the *sending event* task $\sigma_{(v_i,v_j,p)}$ is scheduled just after the task v_i, at time $t = f(v_i, p)$, on processor p. The respective *receiving event* task $\rho_{(v_i,v_j,p')}$ is scheduled on processor p' at time $t' = s(\rho_{(v_i,v_j,p')}, p')$ such that $t' \geq t + \lambda_s + w((v_i, v_j)) * \varphi(p, p')$. The respective tuples $(\sigma_{(v_i,v_j,p)}, p, t)$ and $(\rho_{(v_i,v_j,p')}, p', t')$ are included in S.

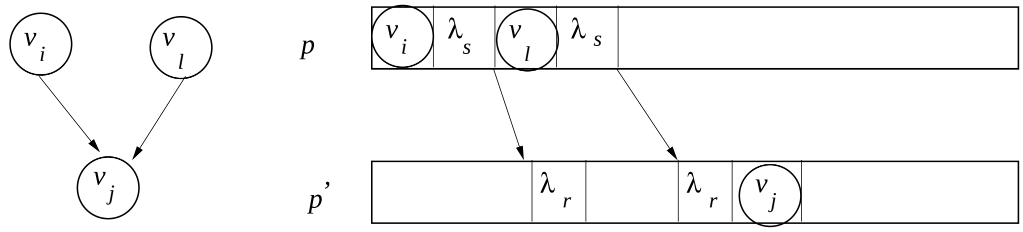


Figure 4.2: A small example of allocation where communication events are incurred when communicating.

Suppose that the other immediate predecessor v_l of v_j in G' is scheduled on the processor p, just after the *sending event* task $\sigma_{(v_i,v_j,p)}$ is executed, i.e. there exits a tuple $(v_l, p, s(v_l, p))$ in S in which $s(v_l, p) = t + \lambda_s$. Consequently, another *sending event* task $\sigma_{(v_l,v_j,p)}$ is scheduled on processor p, i.e. $(\sigma_{(v_l,v_j,p)}, p, s(v_l, p) + m(v_l)) \in S$ and the respective *receiving event* task $\rho_{(v_i,v_j,p')}$ is executed by processor p' when receiving the message (v_l, v_j) at time $s(\rho_{(v_l,v_j,p')}, p') \geq t'' + \lambda_s + w((v_l, v_j)) * \varphi(p, p')$ where $t'' = s(v_l, p) + m(v_l)$.

A better makespan can be achieved if tasks v_i and v_l are *clustered*, in such a way that they are executed sequentially ($s(v_l, p) = s(v_i, p) + m(v_i)$) on the same processor p, and then the messages (v_i, v_j) and (v_l, v_j) are *bundled* and sent to processor p' to enable the execution of task v_j. In this way, only one *sending event* task σ would be scheduled on the processor p at time $s(\sigma, p) \geq \max\{s(v_i, p)+m(v_i), s(v_l, p)+m(v_l)\}$. Also, only one *receiving event* task ρ would be scheduled on processor p' when the *bundled* message reaches this processor, i.e. at time $s(\rho, p') \geq s(\sigma, p)+\lambda_s+W*\varphi(p, p')$, where $W = w((v_i, v_j))+w((v_l, v_j))$ is the resulting size of the *bundled* message sent to processor p'. Therefore, the new resulting makespan of this new scheme would be at least λ_s time units smaller than the makespan of the schedule S.

In the view of the above example, there may be advantages in organising tasks into *clusters*. Each *cluster* is defined in such a way that its tasks are executed consecutively on a single processors. Necessary *communication events* are specified before or after the execution of the cluster so that a smaller number of such events are specified and small messages are bundled.

Definition 4.1.1 *Given a DAG $G = (V, E)$, a cluster of tasks c is a subset of tasks* *in V, listed in their increasing topological order, i.e.*

$$c = \langle v_{i_1}, \ldots, v_{i_{n_c}} \rangle$$

such that $\mathcal{T}_{v_{i_j}} \leq \mathcal{T}_{v_{i_{j+1}}},\ j = 1, \ldots, n_c - 1$

Let $c = \langle v_{i_1}, \ldots, v_{i_{n_c}} \rangle$ denote a cluster of tasks in V allocated to processor $p \in \mathcal{P}$ where $n_c = \mid c \mid$. Let $c' = \langle v_{l_1}, \ldots, v_{l_{n_{c'}}} \rangle$ be another cluster allocated to processor $p' \in \mathcal{P}$ such that there exists a task $v_{i_j} \in c$ where $(v_{i_j}, v_{l_k}) \in E$, for some $v_{l_k} \in c'$. The tuples in S associated with the scheduling of the tasks in clusters c and c' are

$$(v_{i_j}, p, s(v_{i_j}, p)),\ j = 1, \ldots, n_c$$

$$(v_{l_k}, p', s(v_{l_k}, p')),\ k = 1, \ldots, n_{c'}$$

where $s(v_{i_j}, p) = s(v_{i_{j-1}}, p) + m(v_{i_{j-1}}),\ j = 2, \ldots, n_c;\ s(v_{l_k}, p') = s(v_{l_{k-1}}, p') + m(v_{l_{k-1}}),\ k = 2, \ldots, n_{c'}$ and $s(v_{l_1}, p') \geq s(v_{i_{n_c}}, p) + m(v_{i_{n_c}}) + comm$ in which $comm$ is the total communication time, due to the sending and receiving overheads and the delay which are incurred when all of the data between c and c' is transmitted at once between the respective processors p and p' .

With the clusters defined, each tuple in S represents the schedule of each cluster, considering the costs associated with the communication events. One of the problems of finding suitable clusters is that the communication dependencies between clusters must be identified. These dependencies correspond to the communication activities between processors to which the clusters are mapped. Consider the two clusters c and c' defined above. A dependency relation represents the dependency edge between the clusters c and c' due to the precedence relationship (v_{i_j}, v_{l_k}) and the communication between the two respective processors p and p'. This dependency relation and the set of all clusters defined are represented by a directed acyclic graph called a *superdag*, say, $G_{\mathcal{S}} = (V_{\mathcal{S}}, E_{\mathcal{S}})$. The vertices in the superdag, the *supertasks*, correspond to the clusters of tasks and the edges, the so-called *superedges*, are the communications between supertasks, corresponding also to the precedence relationship between them, inherited from the original *DAG* G. The execution cost associated with each supertask $c \in V_{\mathcal{S}}$ is

$$m(c) = \sum_{v_{i_j} \in c} m(v_{i_j})$$

and the weight associated with each superedge $(c, c') \in E_{\mathcal{S}}$ is

$$w((c, c')) \leq \sum_{v_{i_j} \in c, v_{l_k} \in c'} w((v_{i_j}, v_{l_k}))$$

The upper bound on the actual quantity of data transmitted between the clusters c and c' can be explained by the existence of either of the two situations described below. Suppose that a task v_{i_k}, with some immediate successor in cluster c', is replicated:

(a) in cluster c and in another cluster c^* so that the data from v_{i_k} is transmitted through the superedge (c^*, c');

(b) in cluster c and c', so that there is no need for its associated data be transmitted, as it is already available locally to the tasks of cluster c'.

In order to reduce the costs of communication in executing a *DAG,* we propose a *replication* strategy based on the Papadimitriou and Yannakakis' heuristic scheduling algorithm [106], in which each task $v_i \in V$ is clustered together with a certain number of predecessors onto a processor $p \in \mathcal{P}$, in order to bundle small messages into larger ones. It must be borne in mind that, at this stage, the architectural parameters are considered in their *essence* such that there is a need to find a clustering which produces a superdag with granularity that matches the granularity of the architectural model. Each formed cluster, the so-called supertask, is allocated to a distinct *virtual* processor and therefore, an unbounded number of available processors is considered. We shall define a parameter called the *clustering factor* and denoted by γ, which guides the degree of replication, as discussed in details in later sections. This parameter γ is closely related with the structure of the input *DAG* and the granularity of the target system. Although each task v_i has predecessors allocated to the same processor p, messages may be received or sent, if predecessors or successors of tasks in the cluster are not allocated to the same processor p.

Due to the irregularity of the input *DAG*, predecessors of a task $v_j \in V$ may be replicated on a subset of supertasks $\mathcal{C}$. Let $c' \notin \mathcal{C}$ be a supertask containing v_j. Suppose that each supertask in $\mathcal{C}$ contains a copy of v_i, one of the immediate predecessors of v_j, i.e.

$$\exists v_i \in V \ : \ \forall c \in \mathcal{C}, \ v_i \in c \ \wedge \ (v_i, v_j) \in E$$

However, when establishing the dependence relation, just one superedge needs to be defined, as there is no need for $v_j \in c'$ to receive more the one copy of data from the same immediate predecessor v_i. Therefore, one of the clusters $c \in \mathcal{C}$ is selected as the immediate predecessor supertask of c'. In order to select a reasonably efficient candidate supertask in $\mathcal{C}$ that is to send this message, the following constraints are established and combined:

- the number of communication events are the minimal possible so that the overhead is paid the minimal number of times, as it may have a high value;

- the candidate $c \in \mathcal{C}$ selected is the one which sends the largest possible message , i.e.

$$c_{selected} = c \in \mathcal{C} \;:\; \max_{c \in \mathcal{C}} w((c, c'))$$

This selection technique is not necessarily optimal. Both conditions are local and greedy heuristics described in a fuller account in the future Section 5.4.

4.1.1 Related concepts

We can find in the literature models similar to the concept of superdag, however, in some cases, with different purposes. As already discussed in Chapter 2, Papadimitriou and Ullman when proposing the *event* model [105], defined overhead as the relationship between processors allocated to the tasks of a *DAG*. The "edges" among processors represent the precedence relationship inherited from the input *DAG* and the transmission of data. However, this relationship between processors aims to represent message traffic rather than startup overheads.

Anderson *et al.* [7] proposed the *schedule graph*, where tasks are aggregated and each cluster is referred as "jobs". Edges connecting the jobs represent the schedule constraints, i.e. the execution order of the jobs. While a job is being executed, no other event is carried out on the respective processor. The precedence between the jobs represents the time that processors are idle waiting for other jobs to finish, or, the synchronisation cost or even the scheduling costs. Each job comprising t tasks has execution cost equal to t. The scheduling of each job also has unit cost. The makespan of the schedule is the length of the critical path of the *scheduled graph* and the total overhead is the total idle time of the processors added to the number of jobs (scheduling cost).

The main difference between *schedule graph* and our superdag is the objective of the model: in their model, synchronisation cost for shared memory MIMD machines is represented. The precedence between the jobs corresponds the smallest number of edges that are necessary to represent the necessary precedence. In a similar manner, we also wish to define the smallest number of edges (our superedges), because it represent the smallest number of communication events that may be incurred in the parallel execution of the input *DAG*. However, our main objective is not to model synchronisation cost, but to find a good schedule considering the CLAUD model. The supertasks are similar to the jobs, but our aim when clustering tasks is to bundle small messages into larger ones so that the number of communication events is decreased. We note that while our model allows replication of tasks on different supertasks, the *schedule graph* does not define replication of tasks on various jobs. In [13], Bampis *et al.* evaluate the lower and upper bound of the schedule of binary trees and the *diamond DAG*, considering Anderson *et al.* model.

4.2 Remapping to the physical processors

When executing the first stage of the *multi-stage approach*, we can ignore the issue related to the availability of physical processors. At most, as many supertasks as the number of tasks in the original *DAG* are created. When the input *DAG* has a higher number of tasks than the number of available physical processors, a *reallocation* procedure must be performed. The second stage of the *multi-stage approach* is characterised by another parameter of the architecture: the number of processors m. Procedures are employed so that the superdag produced by the first stage can be executed reasonably on m processors.

Definition 4.2.1 *Two supertasks c, c' are independent if the superedges (c, c') and (c', c) are not defined. Let p and p' respectively denote the processors to which c and c' have been assigned in a schedule, say, S. The supertasks are parallel in S, if $p \neq p'$, $s(c, p) \leq s(c', p') < f(c, p)$ or $s(c', p') \leq s(c, p) < f(c', p')$, i.e. they are executed on distinct processors p and p' and their execution times overlap each other.*

Given schedule S and the superdag, if the number of supertasks is greater than m, the supertasks may be clustered through the merging of parallel supertasks and *remapped* to the physical processors. When *remapping* the degree of parallelism is exploited and the supertasks are merged in such a way that the number of communication events is minimised, though an increase in the length of messages may be necessary. With this objective, Brent's Principle [20] is applied together with the proper evaluation of the precedence relationship between the supertasks, resulting in a new superdag in which the number of parallel supertasks executing at *any moment* is less than or equal to m.

The set of parallel supertasks that are executed during a particular period of time are re-scheduled and re-allocated to m processors through the application of Brent's Principle. Therefore, instead of remapping the supertasks to m, the degree of parallelism is maximised by re-scheduling each subset of parallel supertasks to m processors. Let C_i be the set parallel supertasks executed during some period of time. Using Brent's Principle, the number of supertasks reallocated to the same processor is

$$\left\lceil \frac{| C_i |}{m} \right\rceil$$

The selected supertasks in C_i to be reallocated to the same processor are the ones which have the largest number of common immediate successors. Merging supertasks can lead to the bundling of superedges and therefore, the weights associated to the new superedges are re-evaluated. The new supertask also has its execution costs re-evaluated.

Mainly, the rest of this dissertation is dedicated to the description, specification and evaluation of the new *multi-stage approach*, also referred as MSA. Each stage is comprised of more than one heuristic which are discussed in its full detail in the chapters that follows.

Chapter 5

Clustering of Tasks and Communications: The first stage

5.1 Introduction

A scheduling heuristic is intended to define a good schedule for a *DAG* $G = (V, E)$ under the assumptions that characterise the target model. We have already described the characteristics of our target model: communication events and delay are part of the communication costs. A balance between the degree of parallelism of the schedule and the number of communication events must be found so that the input application can be executed efficiently on the parallel architecture.

In this Chapter, the methods that form the first stage of the *multi-stage approach* are described and discussed. This stage consists of a sequence of two heuristics which transform the input *DAG* $G = (V, E)$ into a superdag with characteristics that match the granularity of the target architecture. To achieve this, *replication* of tasks is used to bundle messages through the clustering of tasks, thereby avoiding unnecessary communication events. This technique is necessary when communication overhead is significant in the target architecture. Jung *et al.* [70] analysed the importance of replication considering the Papadimitriou and Yannakakis model. They concluded that the best schedule produced by an algorithm that ignores replication may be worse than the true optimal schedule by a multiplicative factor which is a function of τ.

Note that the first stage does not include the actual architectural *values* and therefore does not produce the final schedule. However, the architectural parameters are considered so that the heuristics can produce a good superdag. Nevertheless, no limit on the number of processors is specified at this stage: there exist as many processors as tasks in G.

5.2 Bundling messages and tasks replication

At first, the *replication* principle that motivated the clustering of tasks in our *multi-stage approach* is described and then, we are ready to discuss the proposed methodology.

5.2.1 Papadimitriou and Yannakakis' heuristic

In [106], Papadimitriou and Yannakakis proposed an algorithm (PY) which generates a schedule with a makespan within a factor of two of the optimal for a given *DAG* $G = (V, E)$ on an unbounded number of processors. Their work only considers one important parameter of the architecture — the message delivery delay τ, defined as the communication cost to computation ratio. The *earliest start time* of a task $v_i \in V$ is the earliest time that v_i can start to be executed taking into account the predecessors of v_i and the communication delay τ and is denoted by $e_{PY}(v_i)$. The main objective of their heuristic is to execute some of the predecessors of v_i on the same processor as v_i in order to hide the cost of communicating with predecessors allocated to other processors. The algorithm schedules each vertex v_i onto a unique processor $p_i \in \mathcal{P}$ at time step $2e_{PY}(v_i)$, together with its τ-highest predecessors (the predecessors of v_i being listed in the decreasing order of their e_{PY}-values). The schedule time of each task v_i is such that v_i and its τ predecessors will be able to receive all of the data necessary for their execution. In summary, assuming an unbounded number of available processors, PY:

- schedules a task v_i on a distinct processor $p_i \in \mathcal{P}$ at time step $2e_{PY}(v_i)$,
- replicates up to the τ-highest predecessors of v_i on the same processor p_i, at the τ time steps previous to $2e_{PY}(v_i)$. If v_i has less than τ predecessors, then all of its predecessors are replicated, and

The e_{PY}-values are such that there is no schedule in which a task is scheduled before its e_{PY}-value and also, the above heuristic produces a schedule within a factor of 2 of the optimum [106]. The tasks are scheduled at time $2e_{PY}(v_i)$ with its τ-highest predecessors so that there is time enough (τ time steps) for the earliest predecessor in the cluster to receive data, if necessary.

5.2.1.1 Reducing the number of messages

In effect, when PY allocates the $\tau + 1$ tasks together on the same processor, clusters of tasks are being formed. The clustering of a task v_i with some of its predecessors reduces the number of communication events since no communication cost is incurred between tasks allocated to the same *virtual* processor. Another strategy that may reduce the number of messages received by a task v_i from predecessors clustered together on another processor is, instead of each predecessor sending a small message, to bundle all of the messages into a larger one, reducing the number of times that the overheads are incurred.

Before any of the tasks within a cluster can be executed, the processor must receive all of the necessary data and only after the processor has executed all of the tasks within the cluster, is the resulting data sent to the respective successors, thus minimising the number of sending and receiving events.

5.2.2 The clustering factor

In MSA, the concept of PY is used but in slightly different manner, as will be described in this section. In MSA, main objective behind the adoption of the *replication* principle is to cluster tasks in order to bundle a number of small messages into fewer, larger ones, since the overheads are also important parameters that must be taken into account when processors communicate. The *replication principle* is the basis of the proposed *replication* algorithm.

Throughout this work, it will be observed that the best number of tasks in a cluster may depend not only on characteristics of the architectural model, e.g. the overheads λ_s and λ_r or just τ as in PY, but also on characteristics of the input *DAG* G, e.g. the in- or out-degree of the tasks in G. The parameter that dictates the number of tasks to be clustered in the *replication* algorithm, is denoted by the *clustering factor* γ.

The value of γ represents the upper limit on the number of tasks to be grouped together, thus forming similarly sized sets of tasks. In fact, the clustering corresponds to an adjustment to the size of the "tasks" (clusters) to improve the ratio of computation and communication times, i.e. to reduce the relative communication costs. This may result in the amount of concurrency being reduced in order to overcome the problem of high communication costs.

5.2.3 Defining the e-function

In order to cluster a subset of tasks of $G = (V, E)$, the earliest start time or e-value of each task $v_i \in V$ is calculated based on the *clustering factor* γ and denoted by $e(v_i)$ in MSA.

The definition of the e-value of the tasks in G follows the precedence relationship specified by E, so that whenever the e-value of v is being calculated, all of its predecessors have already had their e-value assigned. Therefore, the source tasks are the first to be evaluated and as these tasks do not have predecessors, their e-values are set to zero.

Let the set of all predecessors or ancestors of a vertex v_i be denoted by $ancestors(v_i)$. Then, $e(v_i) = 0, \quad$ if $ancestors(v_i) = \emptyset$. Let

$$elist(v_i) = \langle u_0, u_1, \ldots, u_t \rangle \ : \ u_i \in ancestors(v_i), \ e(u_i) \geq e(u_{i+1})$$

be the list of ancestors of v_i in the decreasing order of their e-values, in which $t = \mid ancestors(v_i) \mid$. Then, the e-value of v_i is defined as

$$e(v_i) = \begin{cases} e(u_{\gamma+1}) + \gamma + 1, & \text{if } v_i \text{ has more than } \gamma \text{ ancestors} \\ \mid ancestor(v_i) \mid, & \text{otherwise} \end{cases} \tag{5.1}$$

When v_i has more than γ ancestors, its cluster communicates with the $(\gamma + 1)$th task in $elist(v_i)$, making sure that there is enough time for all tasks inside the cluster to receive data from their respective immediate predecessors allocated to others processors.

Definition 5.2.1 *The cluster formed by the scheduling of a task v_i is called the v_i-cluster and v_i is the owner of it. Let c also denote the v_i-cluster. Then, v_i is the c-owner.*

Since a task v_i can be replicated in more than one cluster, it is useful to determine the set of clusters in which v_i is replicated.

Definition 5.2.2 *$Clusters(v_i, S)$ is the list of all clusters in which the task v_i is scheduled, considering the schedule S produced by the replication algorithm.*

5.2.3.1 Defining computational intervals

With the e-values calculated, the schedule of the tasks can be defined so that each cluster constructed by the *replication* algorithm is scheduled on a single and distinct *virtual* processor. We start the analysis of PY with those tasks that have γ or less ancestors. Each of these tasks is scheduled together with all of its ancestors, so that the resulting clusters do not communicate among themselves. The e-values of these tasks fall within the interval $[0, \gamma]$ and therefore, the owners of their respective clusters are scheduled within the interval $[0, 2\gamma]$, when PY is applied. Let this first group of parallel clusters be denoted by C_0.

The next group of parallel clusters, denoted by C_1, are the ones whose owners have more then γ ancestors, and therefore must communicate with clusters in C_0, i.e. for each cluster $c \in C_1$ there exists at least one $c' \in C_0$ such that c' sends data to c. Therefore, an owner v_i of a cluster $c \in C_1$ must have its e-value within the interval $[\gamma + 1, 2\gamma + 1]$, since $e(v_i) = e(u_{\gamma+1}) + \gamma + 1$, in which $u_{\gamma+1}$ belongs to some cluster $c' \in C_0$. Hence, owners of clusters in C_1 are scheduled within the interval $[2\gamma + 2, 4\gamma + 2]$.

By induction, assume that the set C_i, for $i \geq 2$, contains the clusters whose owners are scheduled within the time interval $[2i * (\gamma + 1), 2(i * (\gamma + 1) + \gamma)]$, since their e-values are within the range $[i * (\gamma + 1), i * (\gamma + 1) + \gamma]$. A cluster $c \in C_i$ with owner v_j communicates with the $(\gamma + 1)$t͟h highest predecessor in the list $elist(v_j)$; let u be this predecessor. Therefore, $e(u) = e(v_j) - (\gamma + 1)$ and so, there exists a cluster whose owner is the task u, which is scheduled within the interval $[2(i - 1) * (\gamma + 1), 2((i - 1) * (\gamma + 1) + \gamma)]$. The set of all clusters scheduled within this interval is the set C_{i-1}.

Definition 5.2.3 *A critical cluster $c \in C_i$ is such that postponing it increases the makespan, and therefore, its owner starts exactly at time step $2(i * (\gamma + 1) + \gamma)$, where C_i is the set of clusters scheduled within interval $[2i * (\gamma + 1), 2(i * (\gamma + 1) + \gamma) + 1]$.*

Lemma 5.2.1 *Let v_j-cluster be a critical cluster with more than γ ancestors. Therefore, there exists a task $u \in V$ such that $u \in ancestors(v_j)$ and $e(v_j) = e(u) + \gamma + 1$. Moreover, u-cluster is also a critical cluster.*

Proof:

It follows from the definition of e-value that the owner v_j of the *critical cluster* $c \in C_i$ has e-value $e(v_j) = e(u) + \gamma + 1$, for some task u which is scheduled at time $2e(u)$, as defined in the PY heuristic. Therefore, the u-cluster is scheduled at time

$$2 * (e(v_j) - (\gamma + 1)) - \gamma = 2((i - 1) * (\gamma + 1) + \gamma) - \gamma$$

which is the latest scheduling time for owner of clusters in C_{i-1}. Thus from Definition 5.2.3, the u-cluster is also a *critical cluster*. $\square$

This shows that *critical cluster* always depends on another *critical cluster* scheduled in a different interval. Hence, the makespan of the schedule S does not increase if the non-*critical clusters* in each interval C_i are postponed in such a way that they finish executing at time step $2(i * (\gamma + 1) + \gamma) + 1$. We modify the definition of the schedule S so that the owners of clusters in each subset C_i are scheduled exactly at time $2(i * (\gamma + 1) + \gamma)$. The aim of this modification is to precisely group independent (parallel) clusters into bands of width $\gamma + 1$. This strategy allows us to identify accurately the parallel clusters, which are executed during *computational intervals*. Between any pair of *computational intervals*, a *communication interval* is defined, within which no tasks from the input *DAG* is executed. The usefulness of this definition will be clarified in future stages of the global transformation scheme, but be aware that the *computational* and *communication* intervals are not meant to be *synchronisation* points.

5.3 The *DAG* structure analyses

The communication among tasks in a *DAG* $G = (V, E)$ is represented by the edges in the set E and therefore, these edges are closely related to communication events. Whenever adjacent tasks are allocated to distinct processors, *sending* and *receiving* tasks are allocated to the communicating processors, respectively.

When an application like the diamond *DAG* is partitioned into lines and the resulting schedule is used to execute the *DAG* on a target architecture where communication cost are very high (overheads or latency), the time to completion is reasonably high [18]. Therefore, scheduling algorithms which produce such partition for the diamond *DAG* should not be applied when considering the CLAUD model. As already defined in Chapter 4, communication between two supertasks of a *superdag* is carried out only after all the tasks inside source *supertask* are executed and all data sent by these tasks is transmitted at once through the respective *superedge*. The use of a superdag to represent the clusters of tasks produced by the *lines* partition and the relationship between the clusters, corresponds to the fact that the lines are executed one after another, producing a schedule with makespan no better than the sequential execution of the *DAG*, considering the case in which each line is allocated to a distinct processors.

In order to overcome these problems, we evaluate the applicability of the boxes partition on the diamond *DAG* considering the CLAUD model. In [104, 105], the benefits of the *boxes* partition on the diamond DAG are discussed when communication events are given relevance in the target model. The diamond *DAG* is a regular structure so the partitioning into *boxes* is straightforward, and a super-diamond *DAG* can be specified and the CLAUD model applied. The important characteristic addressed here is that *boxes* correspond to regular patterns identified on a regular structure like the diamond *DAG*.

Considering *DAG*s in general, i.e. not only regular but also irregular *DAG*s, the *lines* partition could be thought of as the clustering of tasks along the same path. The problems caused by such partitioning are the same as the ones identified when evaluating the *diamond DAG*: a large number of communication events may be incurred in the critical path of the *DAG*. In the next section, a methodology is proposed and discussed so that edges are bundled rather than tasks just being clustered. This methodology is an attempt to avoid clustering schemes like the *lines* partitions which may not lead to a good solution considering a target model where communication events are of great relevance.

5.3.1 Is the e-value sufficient to cluster tasks?

The replication of tasks is guided by the e-function, which considers the ancestors of a task. The selection of ancestors to be clustered follows their decreasing order of the e-value, as listed in $elist(v)$. The clustering of tasks has as its main objective the bundling of messages, so that the number of communication events is minimised. Is the definition of $elist(v)$ sufficient to properly bundle the right edges so that a low number of communication events is incurred?

Let $G = (V, E)$ be a diamond *DAG* and $v \in V$. With the objective to construct a cluster in which the tasks inside the cluster are highly connected, the following mechanism or priority is proposed for the construction of a cluster.

Priority: Initially, v-cluster is empty. Include v in v-cluster. Then, select a predecessor of v with the *highest number of immediate successors* already in v-cluster. This mechanism is repeated until the required number of tasks to be clustered is achieved.

The ordering of inclusions is similar to the *breadth-first search* manner [30], considering the adjacency in the reverse direction. The *highest number of immediate successors* priority is incorporated in the *replication* algorithm which produces a schedule where sets of parallel clusters are defined. In order to cluster tasks with "high connectivity", we combine the clustering of tasks using the e-values and the *highest number of immediate successors* priority. Let $v \in V$

be a task and

$$elist(v) = \langle u_1, \dots, u_i, \dots, u_{\gamma+1}, \dots, u_j, \dots u_t \rangle$$

be the set of t ancestors of v listed in the decreasing order of their e-values. The e-value of v is $e(u_{\gamma+1}) + \gamma + 1$. Suppose that there exists more than one task in $elist(v)$ with the same e-value $e(u_{\gamma+1})$; that is, there exists a subset $\mathcal{A}(v) = \{u_i, \dots, u_{\gamma+1}, \dots, u_j\}$ of tasks in $elist(v)$ such that $e(u) = e(u_{\gamma+1}),\ \forall u \in \mathcal{A}(v)$. In such a situation, any $\gamma - i$ tasks from $\mathcal{A}(v)$ can be selected and incorporated to the v-cluster. In order to break ties on the selection of tasks in $\mathcal{A}(v)$ which have the same priority, the *highest number of immediate successors* priority is applied, forcing the bundling of edges at the "border" of the cluster.

5.3.2 Broadening the clustering space

The scheduling time of a v_i-cluster, for each $v_i \in V$, is defined as the latest time at which it can be executed. Therefore, the e-value of *non-critical clusters* are not used to define their respective scheduling time; only the e-values of the *critical clusters* are used.

In the same manner in which the scheduling time v_i-cluster is defined as the latest possible, regardless of whether it is a *critical cluster*, the e-value of *non-critical-clusters* can also be postponed without affecting the resulting scheduling. We define the *latest start time* $e_l(v)$ as the latest time at which a task v can start without increasing the makespan of the resulting scheduling S. Let the set of all predecessors or ancestors of a vertex v_i be denoted by $ancestors(v_i)$. Let

$$elist_l(v_i) = \langle u_0, u_1, \dots, u_t \rangle \ : \ u_i \in ancestors(v_i),\ e_l(u_i) \geq e_l(u_{i+1})$$

be the list of ancestors of v_i in the decreasing order of their e_l-values, in which $t = \mid ancestors(v_i) \mid -1$. Then the e_l-value of v_i is defined as

$$e_l(v_i) = \begin{cases} e_l(u_{\gamma+1}) + \gamma + 1, & \text{if } v_i \text{ has more than } \gamma \text{ ancestors} \\ \gamma + 1, & \text{otherwise} \end{cases} \tag{5.2}$$

The advantage of using the tasks e_l-values rather than the e-values is the increase of the number of candidate tasks to be clustered when applying the *highest number of immediate successors* priority. For a given task $v_i \in V$, the list $elist_l(v_i) = \langle u_0, u_1, \dots, u_\gamma, u_{\gamma+1}, \dots, u_t \rangle$ is such that the number of tasks with the same $e_l(u_{\gamma+1})$ is greater than or equal to the number of tasks with the same $e(u_{\gamma+1})$. These tasks are γ time units further from v_i and therefore, any of them can be selected to be clustered together with v_i. Finally, like the e-value, the bands of parallel clusters are precisely defined when the e_l-values are also used.

By definition, all of the tasks $v_i \in V$ which are such that $\mid ancestors(v_i) \mid \leq \gamma$, have their e_l-value equal to $\gamma + 1$ and their respective clusters are scheduled during the first computation in S. Those tasks with $\gamma < \mid ancestors(v_i) \mid \leq 2\gamma$, have their e_l-value equal to $2(\gamma + 1)$. Therefore, all of the tasks $v_i \in V$ with $e_l(v_i) = k(\gamma + 1)$ are those which $(k-1)\gamma \; < \mid ancestors(v_i) \mid \leq k\gamma$. Therefore, from this discussion follows the Lemma below:

Lemma 5.3.1 *Given a clustering factor γ and a DAG $G = (V, E)$, the e-values and e_l-values are calculated. If $e(v_i) \geq e(v_j)$, for two tasks $v_i, v_j, \in V$, in which $v_j \in ancestors(v_i)$, then $e_l(v_i) \geq e_l(v_j)$.*

Proof:
By the definitions of $elist(v_i)$ and $elist_l(v_i)$. □

Note that if $v_j \notin ancestors(v_i)$, v_j does not interfere in v_i's schedule. The function e_l increases monotonically, and all e_l-values are multiple of $\gamma + 1$. Therefore, the tasks inside in each cluster have their e_l-values along at most two subsequent multiples of $\gamma + 1$, as shown in the Lemma below.

Lemma 5.3.2 *Let $G = (V, E)$ and $v_i \in V$. Let $e_l(v_i) = k(\gamma + 1)$, for some integer $k > 0$. Then, there exist $x \leq \gamma$ tasks in $elist_l(v_i)$ with their e_l-value greater than $(k-1)(\gamma + 1)$ and smaller than $k(\gamma + 1)$.*

Proof:
Suppose the contrary, i.e. $x > \gamma$. By definition, $elist_l(v_i) = \langle u_1, \ldots, u_\gamma, u_{\gamma+1}, \ldots, u_t \rangle$ where $t = \mid ancestors(v_i) \mid$ and $e_l(u_1) \geq \ldots \geq e_l(u_\gamma) \geq e_l(u_{\gamma+1}) \geq \ldots \geq e_l(u_t)$. Then, $e_l(u_{\gamma+1}) = k(\gamma + 1)$ and consequently, $e_l(v_i) = (k+1)(\gamma + 1)$, which is a contradiction. □

5.3.2.1 Constructing the Clusters

The construction of v_i-cluster is specified as follows. Firstly, the list $elist_l(v_i)$ is defined and $e_l(v_i)$ calculated. If $\mid elist_l(v_i) \mid < \gamma + 1$, then all the ancestors are clustered together with v_i. Otherwise, a carefully selection is necessary. Initially, the only task in v_i-cluster is v_i. All of the tasks in $u \in elist_l(v_i)$ with $e_l(u) > e_l(v_i) - (\gamma + 1)$ are included in v_i-cluster, since they do not change $e_l(v_i)$ and there are no more than γ tasks with such e_l values, as shown in Lemma 5.3.2. If $\mid v_i$-cluster $\mid < \gamma + 1$, it is necessary to include more tasks from $elist_l(v_i)$. These tasks are the ones with the e_l-value equal to $e_l(v_i) - (\gamma + 1)$. However, instead of selecting tasks at random, the *highest number of immediate* *successors* priority is applied. For doing so, let:

$$\mathcal{A}_l(v_i) = \{u \in elist_l(v_i) \ : \ e_l(u) = e_l(v_i) - (\gamma + 1)\}$$

The tasks from $\mathcal{A}_l(v_i)$ to be included in the v_i-cluster are selected one by one, because this selection depends on the relationship with tasks already in v_i-cluster. Let $\mathcal{N} = \mid v_i$-cluster $\mid$. While $\mathcal{N} < \gamma + 1$, the following steps are carried out:

1. the next task to be included must be an immediate predecessor of tasks already v_i-cluster. The *highest number of immediate successors* priority is applied by listing $\mathcal{A}_l(v_i)$ in the decreasing order of the number of immediate successors of a task $u \in \mathcal{A}_l(v_i)$, successors which are already in v_i-cluster. This ordered list associated with $\mathcal{A}_l(v_i)$ is represented by

   $$\begin{aligned} \mathcal{OA}_l(v_i) = \ & \langle u_1, \ldots, u_t \rangle \ : \ u_i \in \mathcal{A}_l(v_i) \wedge \\ & \mid succ(u_1) \ \cap \ v_i\text{-cluster} \mid \geq \ \ldots \ \geq \mid succ(u_t) \ \cap \ v_i\text{-cluster} \mid \end{aligned}$$

   where $t = \mid \mathcal{A}_l(v_i) \mid$.

2. Let $x = \mid succ(u_1) \cap v_i$-cluster $\mid$, such that $u_1 \in \mathcal{OA}_l(v_i)$, i.e. x represents the highest number of immediate successors in v_i-cluster of task in $\mathcal{OA}_l(v_i)$. Let

   $$\mathcal{X} = \{u_i \in \mathcal{OA}_l(v_i) \ : \mid succ(u_i) \ \cap \ v_i\text{-cluster} \mid \neq \ x\}$$

   be the set of tasks in $\mathcal{OA}_l(v_i)$ which have less than x immediate successors in the v_i-cluster. This subset is discarded. Thus,

   $$\mathcal{OA}_l(v_i) = \mathcal{OA}_l(v_i) - \mathcal{X}$$

   If the resulting $\mid \mathcal{OA}_l(v_i) \mid = 1$, then the single task is included in v_i-cluster. Otherwise, the next step is carried out.

3. Let $\mathcal{Y}$ be a subset of tasks in the resulting $\mathcal{OA}_l(v_i)$, each task having all its immediate successors as ancestors of v_i and therefore, these immediate successors are already in v_i-cluster, i.e.

   $$\mathcal{Y} = \{u \in \mathcal{OA}_l(v_i) \ : \ (succ(u) \cap ancestors(v_i)) \ \subseteq \ v_i\text{-cluster}\}$$

   These tasks in $\mathcal{Y}$ must have a higher priority to be included in the v_i-cluster than any other task in $\mathcal{OA}_l(v_i)$, because in this way a higher number of edges are bundled. Consequently, the inclusion a task $u \in \mathcal{Y}$ in v_i-cluster decreases the resulting number of communication events. Therefore, if $\mathcal{Y} \neq \emptyset$, then

   $$\mathcal{OA}_l(v_i) = \mathcal{Y}$$

4. Finally, select the first task in $\mathcal{OA}_l(v_i)$ and $\mathcal{N}$ is updated to $\mathcal{N}+1$.

The steps enumerated are repeated until $\mathcal{N} = \gamma + 1$, when the v_i-cluster is formed.

At last, the *replication* algorithm can be implemented and it is specified in Algorithm 4. For each task v_i in V, the $elist_l(v_i)$ is defined in line 7, so that $e_l(v_i)$ can be determined as in Equation 5.2. Note that, the structure A and Q are used here to assure that when v_i is evaluated, all its ancestors have already been visited, and have their e-values calculated. If v_i has less than $\gamma + 1$ ancestors, all of its ancestors are clustered together with it, regardless of their e_l-value (lines 10 and 11). Otherwise, the procedure $form\_cluster$, which incorporates the *highest number of immediate successors* priority, is applied. The scheduling times are defined in lines 11 and 13 . The v_i-cluster represents the tasks in $c = \langle u_\gamma, \ldots, u_1, v_i \rangle$, where the tasks are ordered under their precedence relationship and e_l-values. From lines 15 to 17, the respective schedule times and allocation of tasks in v_i-cluster (the tasks are all allocated to the same *virtual* processor p_i) are specified. The resulting set of all clusters is defined as $V_{cluster}$ in line 19.

Algorithm: 4 $replication(G = (V, E), \gamma)$ {

1 $S := \emptyset;\ \ Q := \emptyset;$

2 for all $v_i \in V$ do {

3 $\quad A(v_i) := \mid pred(v_i) \mid;$

4 $\quad$ if $(A(v_i) = 0)\ \ Q := Q \cup \{v_i\};$

}

5 while $(Q \neq \emptyset)$ do {

6 $\quad t := \mid ancestors(v_i) \mid;$

7 $\quad elist_l(v_i) := \langle u_0, u_1, \ldots u_t \rangle \ :\ u_j \in ancestors(v_i),\ e_l(u_1) \geq \ldots \geq e_l(u_t);$

8 $\quad$ determine $e_l(v_i);$

9 $\quad$ if $(t \leq \gamma)$ {

10 $\quad\quad c := elist_l(v_i);$

11 $\quad\quad stime := 2\gamma;$

$\quad$ } else {

12 $\quad\quad c := form\_cluster(v_i, elist_l(v_i));$

13 $\quad\quad stime := s(u_{\gamma+1}) + 2\gamma + 1;$

$\quad$ }

14 $\quad j := 0;$

15 $\quad$ for each $(u_j \in c)$ {

$S := S \cup (u_j, p_i, stime - j);$

$j := j + 1;$

$\}$

$S := S \cup (v_i, p_i, stime);$

$V_{cluster} := V_{cluster} \cup \{c\};$

$\forall u \in succ(v_i)$ do $\{$

$A(u) \ := \ A(u) - 1;$

if $(A(u) = 0)$ $Q := Q \cup \{u\};$

$\}$

$Q := Q - \{v_i\};$

$\}$

$\}$

The procedure $form\_cluster$ is specified in Procedure 5. Given v_i and its $elist_l(v_i)$, the procedure $form\_cluster$ constructs v_i-cluster. In lines 2 and 3, all the tasks in $elist_l(v_i)$ with e_l-values greater than $e_l(u_{\gamma+1})$ are included in the v_i-cluster, which already contains the task v_i, as specified in line 1. While the number of tasks in the v_i cluster is smaller than $\gamma + 1$, lines from 8 to 15 are executed so that a task $u_j \in \mathcal{A}_l(v_i)$ is selected since this selection corresponds to the highest number of edges bundled. In line 14, u_1 is the selected tasks and it is incorporated to the v_i-cluster under the precedence relationship order.

Procedure: 5 $form\_cluster(v_i, elist_l(v_i))$ $\{$

1 $c := \{v_i\}$

2 for each $u_j \in elist_l(v_i)$ such that $(e_l(u_j) > e_l(v_i) - (\gamma + 1))$ do

3 $c := c \cup \{u_j\}$

4 if $(| \, c \, | < \gamma + 1)$ $\{$

5 $\mathcal{A}_l(v_i) := \{u_j \in elist_l(v_i) \ : \ e_l(u_j) = e_l(v_i) - (\gamma + 1)\};$

6 $\mathcal{N} \ := | \, c \, |;$

7 while $(\mathcal{N} < \gamma + 1)$ $\{$

8 $\mathcal{OA}_l(v_i) \ := \ \langle u_1, \ldots, u_t \rangle \ : \ u_j \in \mathcal{A}_l(v_i) \wedge | \, succ(u_1) \ \cap \ c \, | \geq \ \ldots \ \geq | \, succ(u_t) \ \cap \ c \, | \, ;$

9 $x \ := | \, succ(u_1) \cap c \, |;$

10 $\mathcal{X} := \{u \in \mathcal{OA}_l(v_i) \ : | \, succ(u) \ \cap \ c \, | \neq \ x\};$

11 $\mathcal{OA}_l(v_i) := \mathcal{OA}_l(v_i) - \mathcal{X};$

12 $\mathcal{Y} := \{u_j \in \mathcal{OA}_l(v_i) \ : \ (succ(u_j) \cap ancestors(v_i)) \ \subseteq \ c \, \};$

```
            if (𝒴 ≠ ∅)  𝒪𝒜_l(v_i) := 𝒴;
            c := c ∪ {u_1} such that u_1 ∈ 𝒪𝒜_l(v_i);
            𝒩 := 𝒩 + 1;
        }
    }
    return c;
}
```

The cluster c represents the v_i-cluster, which is characterised by the scheduling of v_i on processor p_i together with the γ highest ancestors according to their e_l-value. The result of the *replication* algorithm is a set of clusters of tasks defined by the schedule S, in which each cluster corresponds to a *virtual* processor.

The resulting makespan of the schedule S is denoted by $\mathcal{M}(S)$. During the interval of time $[0, \mathcal{M}(S)]$, there exists

$$\mathcal{B} = \left\lceil \frac{\mathcal{M}(S) - (2\gamma + 1)}{(2\gamma + 1)} \right\rceil$$

sets of parallel clusters, which are executed during *computational intervals* or *bands*, with the owners of each cluster being scheduled at time

$$2(i * (\gamma + 1) + \gamma), \quad 1 \leq i \leq \mathcal{B}$$

Between any two *bands* there exists a *communication interval* where no task of the input *DAG* is scheduled.

5.3.2.2 A small example

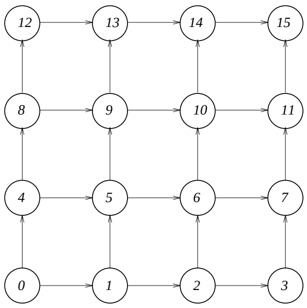


Figure 5.1: 4×4 diamond *DAG*

A diamond *DAG* with dimensions $n \times n$ is used as the input application. In this example, $n = 4$ as shown in Figure 5.1. The tasks are labelled with the subscripts of v_i, starting from the source task, labelled with 0, to the sink, labelled 15. Let $\gamma = 3$ be also given as input.

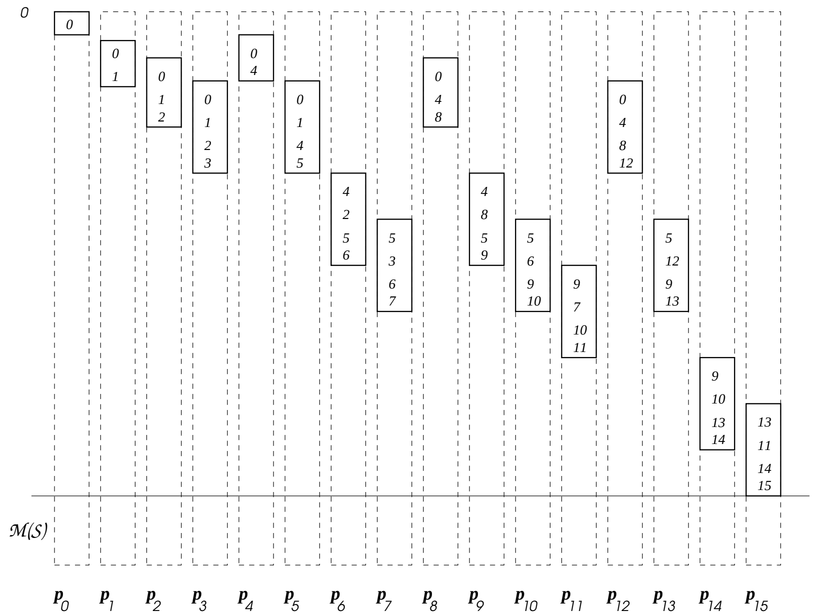


Figure 5.2: Papadimitriou and Yannakakis schedule. Input: diamond *DAG* with $n^2 = 16$ tasks and $\gamma = \tau = 3$

The application of PY to this example results in the schedule S shown in Figure 5.2. The first group of parallel clusters are the ones with owners 0, 1, 2, 3, 4, 5, 8 and 12. The second group contains the clusters with owners 6, 7, 9, 10 and 13 and finally, the last group are the clusters whose owners are 11, 14 and 15. However, in this schedule, the parallel clusters are not clearly defined. The *replication* algorithm overcomes this problem, the Gantt Chart in Figure 5.3 showing that bands of parallel clusters are well defined.

5.3.3 An evaluation of the clusters and intervals

Although the *replication* algorithm constructs clusters of tasks, there is no specification of the communication between them. This communication, or more precisely, the relationship between the clusters will be defined by another heuristic, which is described in future sections and for doing so, some characteristics are evaluated in the following Lemmas. Note that, although individual tasks can be replicated on a number of clusters, each cluster is unique (each owner has a distinct owner) and so it can never be replicated onto two or more processors.

In order to simplify the notation, the schedule time of the v_i-cluster in the set

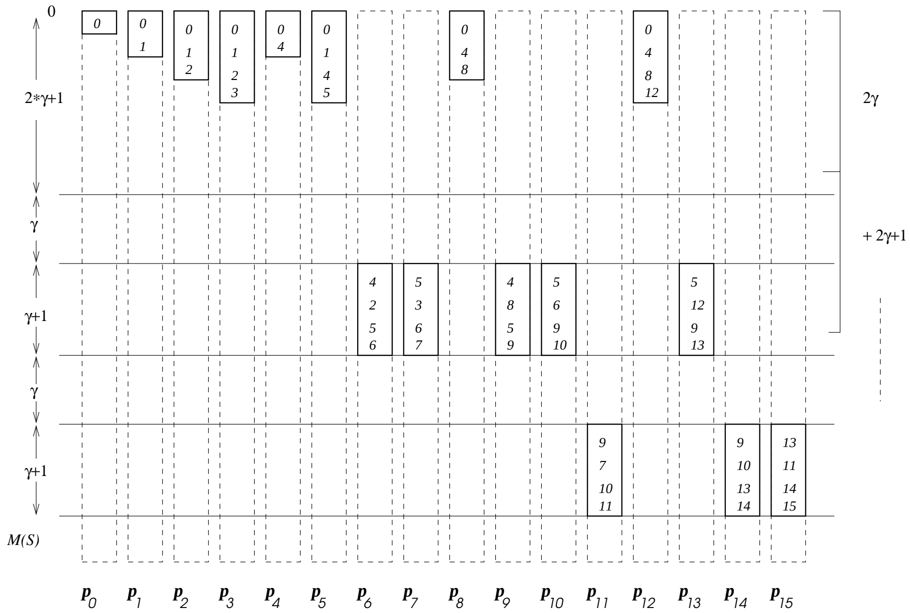


Figure 5.3: Schedule produced by *replication* algorithm on a diamond *DAG* with $n^2 = 16$ tasks and $\gamma = 3$

$V_{cluster}$ (the set of clusters produced by the *replication* algorithm) onto a *virtual* processor p_i is denoted by $s_{cluster}(v_i)$.

Lemma 5.3.3 *Given a DAG* $G = (V, E)$ *and a clustering factor* γ*, let* S *be the schedule produced by the replication algorithm. A task* $u_i \in V$ *can be replicated in cluster which are scheduled in at most two computational intervals in* S*. Furthermore, those two intervals must be adjacent.*

Proof:
Suppose the task u_i is scheduled during two or more distinct computational intervals. Let $v_1, v_3 \in Clusters(u_i, S)$ such that the v_1-cluster is scheduled at the earliest interval, say, I_1, considering the interval of any cluster in $Cluster(u_i, S)$, and v_3-cluster is scheduled at the latest interval of any cluster in $Cluster(u_i, S)$, namely, I_3. By definition, since $u_i \in v_3$-cluster, there exists a task, say, $t_{\gamma+1}$, in $elist(v_3)$ such that $e_l(v_3) = e_l(t_{\gamma+1}) + \gamma + 1$ and $e_l(u_i) \geq e_l(t_{\gamma+1})$ (from Equation 5.2). Consequently,

$$e_l(v_3) \leq e_l(u_i) + \gamma + 1 \tag{5.3}$$

Suppose as hypothesis to be proven contradictory, I_1 and I_3 are non-adjacent intervals, i.e. there is at least one interval that occurs after I_1 and before I_3. In this case, the *replication* algorithm defines the schedule time of v_3-cluster is at least

$$s_{cluster}(v_3) \geq s_{cluster}(v_1) + 4\gamma + 2$$

Since $s_{cluster}(v_3) = 2e_l(v_3) - \gamma$ and $s_{cluster}(v_1) = 2e_l(v_1) - \gamma$, the above equation is reduced to

$$e_l(v_3) \geq e_l(v_1) + 2\gamma + 1$$

Combining the above result with Equation 5.3, we have that $e_l(v_1) \leq e_l(u_i) - \gamma$. However, as $u_i \in v_1$-cluster, then $e_l(v_1) \geq e_l(u_i)$ leading to a contradiction. Clearly, if there were more than two intervals in which u_i was scheduled, then two of these intervals must be non-adjacent. Thus, we have proved that any two intervals in which u_i is scheduled must be adjacent.

□

Corollary 5.3.1 *Given the DAG $G = (V, E)$ and a clustering factor γ, let S be the schedule provided by the replication algorithm. Let $u_i \in V$ be a task with $e(u_i)$, its e-value defined by the e_l-function. Then, no copy of the task u_i is scheduled before $2e_l(u_i) - \gamma$ or after $2e_l(u_i) + 2\gamma + 2$.*

Proof:

Let $v_1, v_2 \in Cluster(u_i, S)$ such that the v_1-cluster is scheduled at the earliest interval of any cluster $Cluster(u_i, S)$ and the v_2-cluster, is scheduled at the latest interval of any cluster in $Cluster(u_i, S)$.

The start time of the v_1-cluster computation interval is $s_{cluster}(v_1) = 2e_l(v_1) - \gamma$, as defined the *replication* algorithm. Since $u_i \in v_1$-cluster, by the definition of the e_l-function (Equation 5.2), $e_l(v_1) \geq e_l(u_i)$ and consequently, $2e_l(v_1) - \gamma \geq 2e_l(u_i) - \gamma$. Therefore,

$$s_{cluster}(v_1) \geq 2e_l(u_i) - \gamma$$

From Lemma 5.3.3, if u_i is executed during two intervals, they are adjacent and $s_{cluster}(v_2) = s_{cluster}(v_1) + 2\gamma + 1$. The finish time of the v_2-cluster computational interval is $s_{cluster}(v_2) + \gamma + 1$, which is no more than

$$2e_l(u_i) + 2\gamma + 2$$

□

5.3.4 A Complexity analysis of the *replication* algorithm

Papadimitriou and Yannakakis proposed the *replication principle* which can be implemented in polynomial time. The *replication* algorithm proposed in this thesis has some additional features which are mainly implemented in the procedure $form\_cluster$.

The loop in line 2 of procedure $form\_cluster$ has at most $\mathcal{O}(\gamma)$ steps. Lemma 5.3.2 says that there exists $x \leq \gamma$ tasks in $elist_l(v_i)$ such that their e_l-values are between $(k-1)(\gamma+1)$ and $k(\gamma+1)$ where $e_l(v_i) = k(\gamma+1)$. The same holds for tasks with e_l-values within the interval $(k-2)(\gamma+1)$ and $(k-1)(\gamma+1)$. Therefore the list $\mathcal{A}(v_i)$, in line 5, has $\mathcal{O}(\gamma)$ elements. Line 7 characterises a loop that has no more than $\mathcal{O}(\gamma)$ iterations. In practical terms, this loop usually has less than γ iterations due to the characteristics of the e_l-function stated in Lemma 5.3.2. Lines 9 to 15, with the exception of line 10, are $\mathcal{O}(1)$. Line 10, on the other hand, has complexity $\mathcal{O}(\gamma)$. Concluding, the complexity of the procedure $form\_cluster$ is $\mathcal{O}(\gamma^2 \log \gamma)$.

The complexity analysis of the *replication* algorithm is given as follows. Line 5 characterises the main loop where each cluster is constructed and takes $\mathcal{O}(n)$ iterations to complete. For each task, the list $elist_l$ is ordered (in line 7) within $\mathcal{O}(n \log n)$ steps. In practice, there may be no need to inspect all of the ancestors of a task v_i. Lemma 5.3.2 states that the e_l-values of the ancestors of v_i are multiples of $\gamma + 1$. Thus, having defined $e_l(v_i)$ as $k(\gamma+1)$, the ancestors included in $elist_l(v_i)$ are those with e_l-values equal to $k(\gamma+1)$, $(k-1)(\gamma+1)$ and, just for security, the ancestors whose e_l-values are equal to $(k-2)(\gamma+1)$ are also considered. Therefore, the construction of $elist_l(v_i)$ can finish when a task $u \in ancestors(v_i)$ with $e_l(u) = (k-2)(\gamma+1)$ is found. For certain, all of the remaining ancestors have e_l-values equal to or less than $e_l(u)$ due to their precedence relationship and also, there will be no more than 2γ ancestors of v_i listed in $elist_l(v_i)$. To conclude, $elist_l(v_i)$ can be implemented in $\mathcal{O}(2\gamma! \log 2\gamma)$ steps. The procedure $form\_cluster$ is executed, in line 12, in $\mathcal{O}(\gamma^2 \log \gamma)$ as analysed earlier. The loop in line 15 has $\mathcal{O}(\gamma)$ iterations and the loop in line 20 has no more than $\mathcal{O}(n)$ iterations. Therefore, the *replication* algorithm can be implemented in $\mathcal{O}(n \max\{\mathcal{O}(n \log n), \mathcal{O}(\gamma^2 \log \gamma)\})$. In practical terms, this complexity can be simplified to $\mathcal{O}(n\gamma^2 \log \gamma)$ for $\gamma \geq 3$.

5.4 Intra-cluster communication

One of the main objective of the clustering of tasks performed in the first stage of the *multi-stage approach* is the bundling of edges of a given *DAG* $G = (V, E)$ so that the cost associated with the sending and receiving of messages is reduced. The schedule S produced by the *replication* algorithm corresponds to the schedule of $\mid V \mid$ clusters, each with at most $\gamma + 1$ tasks. Each cluster is allocated to a distinct *virtual* processor. Although clusters are defined, there is no explicit definition of communication between them.

In later stages of the *multi-stage approach*, transformations will take place which take into account other parameters of the architecture, for example, the number of physical processors in the target architecture. If the number of clusters is much greater than the number of processors, reallocation of distinct clusters to the same processor may be necessary. One must be aware of the assumptions of the problem in order to decide which clusters should be grouped on the same processor, forming a new cluster of tasks. The reallocation procedure must take into account that the overheads λ_s and λ_r may have critical values, and a strategy should be provided so that the number of communication events is minimised. With the reallocation of clusters to the same processor, messages are bundled in order to avoid an increase in the degree (in degree and out degree) of the new clusters.

Due to the needs discussed above, the first stage of the *multi-stage approach* is finalised with the construction of a directed acyclic graph, called a *superdag*, in which the clusters are represented by *supertasks* and the communication, by *superedges*. This allows an explicit representation of the communication between clusters. The algorithm which defines such a *DAG* is called the $superdag$ algorithm and before showing the full procedure, important concepts and mechanisms are discussed.

5.4.1 Defining the immediate predecessors set

Let v_{i_j} be a task which belongs to a cluster, say, c. Due to the *replication* algorithm, there may be a number of clusters which contain an immediate predecessor of v_{i_j}, which does not belong to c. There is no need for v_{i_j} to receive the data from the same immediate predecessor more than once. Therefore, mechanisms are specified in the $superdag$ algorithm in order to choose a cluster to be a "good" *immediate predecessor cluster* of c. For this purpose, it is necessary to know from which tasks the cluster c must receive data:

Definition 5.4.1 *Let $c = \langle v_{i_1}, \ldots, v_{i_{|c|}} \rangle$ be a cluster of $\mid c \mid$ tasks. The set of tasks which do not belong to c and which are immediate predecessors of tasks $v_{i_j} \in c$ are listed in the set $\mathcal{I}(c)$:*

$$\mathcal{I}(c) = \{u_k \in V \;:\; (u_k, v_{i_j}) \in E \;\wedge\; u_k \notin c \;\wedge\; v_{i_j} \in c\}$$

This set of tasks is evaluated by the proposed algorithm. Let $\mathcal{I}'(c)$ be a subset of tasks in $\mathcal{I}(c)$. Initially, $\mathcal{I}'(c) = \mathcal{I}(c)$. The need for the definition of $\mathcal{I}'(c)$ is due to the fact that tasks are removed from it at each iteration, as seen in the description of the algorithm.

5.4.2 Order of the *visits*

In order to define the superedges corresponding to the sending of messages to a cluster c, this cluster has to be *visited*. The order in which the clusters are visited is from the latest clusters to the earliest ones, i.e the visits are done in the decreasing order of the cluster's start time, $s(c, p)$, defined in S, where $(c, p, s(c, p)) \in S$. Let c and c' be two clusters and denote c-owner and c'-owner as v_i and v_j, respectively. Let the processors p and p' allocate c and c'. In order to define the order of the visits, the following Lemma is established:

Lemma 5.4.1 *The topological order of tasks in V, $\mathcal{T}(v)$, is the shortest path from a unique source of the DAG $G = (V, E)$ to v. Given the schedule S produced by the replication algorithm, for any $v_i, v_j \in V$, if $\mathcal{T}(v_i) \geq \mathcal{T}(v_j)$ and $v_j \in ancestors(v_i)$ then*

$$s(c, p) \geq s(c', p')$$

where c corresponds to the v_i-cluster and c' to the v_j-cluster and p and p' are the processors in which the clusters are allocated, respectively.

Proof:
Case 1) $\mid ancestors(v_i) \mid \leq \ \gamma{+}1$: The *DAG* G has a unique source, which can be a dummy source in the case that G contains more than one source task. Then, $elist_l(v_i) = \langle u_1, u_2, \ldots, u_l \rangle$, where $l \leq \gamma + 1$, and v_j is one of the $u_k \in elist_l(v_i)$. Therefore, using the definition of the e_l-values, $e_l(v_j) \leq e_l(v_i)$ and consequently, their scheduling times are $2e_l(v_j) \leq 2e_l(v_i)$.

Case 2) $\mid ancestors(v_i) \mid > \ \gamma{+}1$: Let $elist_l(v_i) = \langle u_1, u_2, \ldots, u_\gamma, u_{\gamma+1}, \ldots, u_l \rangle$. Let $v_k = u_k$ for some $u_k \in elist_l(v_i)$. Two cases are considered:

1) $k \leq \gamma$**:** using the e_l-value definition, $e_l(v_j) \leq e_l(v_i)$.

2) $k > \gamma$**:** using the e_l-value definition, $e_l(v_i) - \gamma + 1 = e_l(u_{\gamma+1})$. Then, by the definition of $elist_l(v_i)$, $e_l(v_i) - \gamma + 1 = e_l(u_{\gamma+1}) \geq e_l(v_j)$. Therefore, $e_l(v_j) \leq e_l(v_i)$.

Concluding, in either case, $s(c, p) \geq s(c', p')$.

□

Corollary 5.4.1 *If there exist two tasks $v_i, v_j \in V$, such $\mathcal{T}(v_i) \geq \mathcal{T}(v_j)$ and $v_j \notin ancestors(v_i)$ then the order of visits is arbitrary.*

Proof:
Since G is a *DAG*, there exists a common task, say u_k on paths from both v_j and v_i to a unique source (even if it is a dummy source). Let c^* represent the u_k-cluster allocated to the *virtual* processor p^* in S. Then, $s(c,p) \geq s(c^*,p^*)$. Also, $s(c',p') \geq s(c^*,p^*)$. Then, both clusters, c and c', are executed before c^* in any order. $\square$

The topological order of the owners is the characteristic that dictates the order in which the clusters are visited. The first visited clusters are the sink ones, clusters whose owners are sink tasks in $G = (V, E)$.

5.4.3 An initial classification of the clusters

Let C_{last} be the set of parallel clusters specified by the *replication* algorithm which finish at time $\mathcal{M}(S)$. The sink tasks from the original *DAG* $G = (V, E)$ are within clusters in C_{last}. However, C_{last} may also contain clusters without sink tasks, which do not need to be executed, as no immediate successors of tasks within them are executed in the future. It is clear to see that these clusters can be eliminated from the resulting superdag. They are referred to *useless* clusters. We know that the sink clusters must remain in the produced superdag.

Definition 5.4.2 *A cluster produced by the replication algorithm is said to be useless, if it is not a sink cluster and it is not an immediate predecessor of any other cluster.*

This characteristic associated with a cluster is a temporal one. As soon as a *useless* cluster is selected to be an immediate predecessor of another cluster, it is not *useless* anymore. There is no need to visit clusters which are not immediate predecessors of clusters already visited. When the $superdag$ algorithm finishes, the clusters which are kept in the resulting superdag to form the set of *supertasks*, are the *non-useless* ones.

The first clusters to be visited are the ones whose owners are sink tasks in the original *DAG* $G = (V, E)$. These clusters are initially given in the decreasing topological order of their owners in a list denoted by C. Therefore, initially,

$$\begin{aligned} C = \ & \langle c_1, \ldots, c_k \rangle \ : \ c_j\text{-owner } \in V \ \wedge \ succ(c_j\text{-owner }) = \emptyset, \ j = 1, \ldots, k \\ & \wedge \mathcal{T}(c_1\text{-owner }) \geq \ldots \geq \mathcal{T}(c_k\text{-owner }) \end{aligned}$$

5.4.4 The main objective of the $superdag$ algorithm

Given the clusters produced by the *replication* algorithm, the main objective of the $superdag$ algorithm is to specify the superedges, and thus construct the superdag, such that the degree of the resulting supertasks may be reduced. If this objective can be achieved, when the resulting superdag is executed in parallel, the number of communication events may be reduced. Minimising the degree of supertasks is essential for the effectiveness of the scheduling approach under the CLAUD model. In order to achieve this objective, the $superdag$ algorithm implements a set of rules which attempt to define the required superdag with a granularity that matches the granularity of the target architecture. These rules are applied iteratively, starting from the sink clusters and selecting the best possible immediate predecessors.

The $superdag$ algorithm visits each *non-useless* cluster c in order to define the necessary superedges into c. Each *visit* performed by the $superdag$ algorithm is called a *phase* of the algorithm. Let $phase_c$ be the phase during which the supertask c is visited. Consider $phase_c$. When defining a superedge, say (c', c), the data from tasks in $\mathcal{I}'(c)$ which belong to c' can be obtained via this superedge, so that the sending tasks do not need to be considered again and can be removed from $\mathcal{I}'(c)$ (since the definition of the superedge (c', c) corresponds to the fact that the data from such tasks are sent through (c', c)). Therefore, while there exist tasks in $\mathcal{I}'(c)$, superedges still need to be defined. At each time that it is verified that $\mathcal{I}'(c)$ is not empty, an *iteration* of the $superdag$ algorithm is initiated, and we shall see that, at each iteration, a superedge is defined. Therefore, the $superdag$ algorithm is composed of as many phases as there are clusters to be visited (the upper bound is equal to $\mid V \mid = \mid V_{cluster} \mid$ phases), and each phase corresponds to a sequence of iterations (no more than $\mid \mathcal{I}'(c) \mid$ iterations).

A set of *rules* used to select a reasonably “good” immediate predecessor cluster, is now described. It must be borne in mind that the rules are established taking into account the main objective of the proposed strategy. Therefore, we shall see that the rules are applied in order to decrease the number of communication events (so that the communication cost is minimised) and at the same time an attempt to minimise the computation cost is performed. Briefly, the rules are listed as follows:

Rule 1) Candidate Clusters Set - defines as candidate clusters the ones finishing their execution by time $s(c, p) - \gamma$;

Rule 2) Restricting the Set - decreases the search space by keeping the candidates which contain immediate predecessor tasks that are less replicated;

Rule 3) Unique Copy - checks if there exists a cluster containing a unique replication of an immediate predecessor task, and establishes such clusters as immediate predecessor clusters;

Rule 4) Maximal Subset Choice - decreases the search space by keeping candidates with a maximal-cardinality subset of the set of predecessor tasks;

Rule 5) Degree - decreases the search space by keeping candidate clusters with the minimum out degree;

Rule 6) Same Interval - maintains the latest candidate clusters considering their schedule time in S;

Rule 7) Topological Order - finally, chooses the one amongst those candidates whose respective owners have the smallest topological order.

Next, the rules are discussed in more details.

5.4.4.1 Rule 1: Listing candidates predecessor clusters

Let c be a cluster and $\mathcal{I}'(c)$ as established in Definition 5.4.1. Let $s(c,p)$ be the start time of the cluster c executed on a *virtual* processor p and $f(c,p)$, its finish time, both times specified in the schedule S. A cluster c' is an immediate predecessor candidate cluster of c if:

1) its execution finishes by time $s(c,p)-\gamma$;
2) it contains at least one of the tasks in $\mathcal{I}'(c)$.

The set of *candidate clusters* of a visited cluster c, denoted by $\Delta(c)$, is the set of clusters in which both of the above conditions holds, i.e

$$\Delta(c) = \{c' \in V_{cluster} \ : \ s(c',p') \leq s(c,p) - \gamma \ \wedge \ \exists u_j \in c' \text{ such that } u_j \in \mathcal{I}'(c)\}$$

where $V_{cluster}$ is the set of clusters produced by the *replication* algorithm.

5.4.4.2 Rule 2: Restricting the number of candidates

This rule attempts to restrict the search space during an iteration. Note that, it is not always the case that a superedge is defined after the execution of this rule and therefore, further rules are necessary. The maximal size of the set $\Delta(c)$ is bounded by the cardinality of the power set of $\mathcal{I}'(c)$, not including the empty set ($\emptyset$). It might not be cost-effective to evaluate all of these candidate clusters. Restrictions should be applied to this set so that only the candidates which have most chance of being selected as immediate predecessors are kept in the search space.

A small example can show how the number of candidate clusters in $\Delta(c)$ can be decreased in a given iteration. In the situation shown in Figure 5.4, in which $\mathcal{I}'(c) = \{u_1, u_2, u_3\}$ is a set of tasks allocated to clusters in $\Delta(c)$, there exists 1 cluster containing the task u_1, 3 clusters containing u_2 and 1 clusters containing u_3. The tasks u_1 and u_3 are the ones that have been replicated least. As the data from u_1 and u_3 must be available to c, the clusters to which such tasks are allocated should be first evaluated, because these clusters have a higher probability of being selected as immediate predecessors during this iteration than the other candidate clusters. Several iterations may be necessary in order to define all the required superedges, but at each iteration, a restricted search space will provide a more efficient evaluation mechanism for superedge specification.

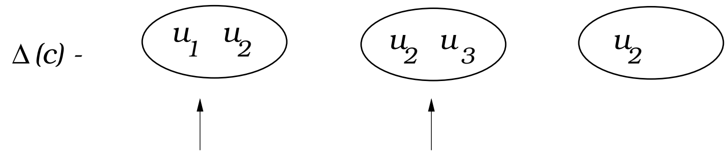


Figure 5.4: Immediate predecessors of tasks in the cluster c are $\mathcal{I}'(c) = \{u_1, u_2, u_3\}$. The arrows shows the restrict search space.

In summary, this rule can decrease the number of candidate clusters to be evaluated in a given iteration by selecting those clusters in $\Delta(c)$ to which the least replicated tasks have been allocated. The number of clusters in which each task $u_i \in \mathcal{I}'(c)$ is allocated is denoted by:

$$Ncluster(u_i, c) = \sum_{c' \in \Delta(c)} \mid \{u_i\} \cap c' \mid$$

Thus, the *restricted* set of candidate clusters, denoted by $\Delta'(c)$ is:

$$\Delta'(c) = \{c' \in \Delta(c) \mid u_i \in c' \ \wedge \ Ncluster(u_i, c) = y\}$$

given that $y = \min_{\forall u_i \in \mathcal{I}'(c)} \{Ncluster(u_i, c)\}$.

5.4.4.3 Rule 3: Unique choice

If there exists an immediate predecessor $u_i \in \mathcal{I}'(c)$ which belongs to only one of the clusters $c' \in \Delta'(c)$, i.e. $\exists u_i \in \mathcal{I}'(c)$ such that $Ncluster(u_i, c) = 1$, then c' must be chosen as one of the immediate predecessor clusters of c. For all tasks $u_i \in \mathcal{I}'(c)$ such that $Ncluster(u_i, c) = 1$, a superedge can be added from the cluster c' containing u_i, to the cluster c. Suppose that c' also contains another task $u_j \in \mathcal{I}'(c)$, where $Ncluster(u_j, c) > 1$. Although c' was selected due to u_i, c' also sends the data corresponding to the task u_j. Therefore, both tasks, u_i and u_j, must be removed from $\mathcal{I}'(c)$ before the next iteration is initiated.

5.4.4.4 Rule 4: Subset choice

If there are still predecessors in $\mathcal{I}'(c)$ and more than one candidate cluster in $\Delta'(c)$, higher priority is given to the cluster $c' \in \Delta'(c)$ which contains the maximum number of tasks in $\mathcal{I}'(c)$. The reason for choosing the cluster with the largest number of immediate predecessors in $\mathcal{I}'(c)$ is to bundle small messages onto a single one, which corresponds to the sending of a long message. Selecting a cluster containing most of the tasks in $\mathcal{I}'(c)$ (ideally, all of the tasks in $\mathcal{I}'(c)$) corresponds to an effort to define a low number of necessary superedges towards cluster c.

Lemma 5.4.2 *Given the current sets $\Delta'(c)$ and $\mathcal{I}'(c)$, the selection of the candidate cluster c' containing the maximal-cardinality subset of $\mathcal{I}'(c)$ leads to the definition of a smaller (although not necessarily the smallest) number of superedges.*

Proof:
The Lemma is shown by analysing extreme situations. Suppose that the set $\Delta'(c)$ contains $\mid \mathcal{I}'(c) \mid$ clusters, each of which contains a single task $u_i \in \mathcal{I}'(c)$, and also, one cluster containing just two tasks, say $u_i, u_j \in \mathcal{I}'(c)$, denoted by c''. Two cases can be considered:

- each of the clusters containing a single task in $\mathcal{I}'(c)$ is selected, resulting on a total of $\mid \mathcal{I}'(c) \mid$ defined superedges;
- the cluster c'' is selected, together with the clusters containing tasks $u_k \in \mathcal{I}'(c)$ such that $u_k \neq u_i$ and $u_k \neq u_j$. Therefore, a total of $\mid \mathcal{I}'(c) \mid -1$ superedges are produced.

Therefore, the selection of the candidate cluster with the maximal-cardinality subset of the set $\mathcal{I}'(c)$ leads to the minimisation of the resulting superedges.

□

In order to find such clusters, the number of immediate predecessors in $\mathcal{I}'(c)$ for each cluster $c' \in \Delta'(c)$ is calculated and denoted as $ntasks(c', c)$, i.e. $ntasks(c', c) = \mid c' \cap \mathcal{I}'(c) \mid$. Let $x = \max_{\forall c' \in \Delta'(c)} \{ntasks(c', c)\}$. Then, the clusters kept in the set $\Delta'(c)$ are those that contain x tasks from $\mathcal{I}'(c)$. The algorithm constructs a subset, denoted by X, of candidate clusters, each of which has less than x tasks from $\mathcal{I}'(c)$:

$$X = \{c' \in \Delta'(c) \ : \ ntasks(c', c) \neq x\}$$

Then, this subset of tasks is eliminated from $\Delta'(c)$:

$$\Delta'(c) = \Delta'(c) - X$$

All of the clusters in the resulting $\Delta'(c)$ contain the same maximum number of tasks x. If $\mid \Delta'(c) \mid = 1$, then an immediate predecessor cluster $c' \in \Delta'(c)$ is found. Otherwise, as more than one candidate clusters still remains further refinements of the search space are carried out.

5.4.4.5 Rule 5: Smallest out degree candidates

Note that, at the current phase of the $superdag$ algorithm, there may exist candidate clusters which are already immediate predecessor clusters, selected in previous phases. Therefore, to break ties among the remaining candidates in $\Delta'(c)$, an evaluation of their out degree is carried out. The action of selecting candidates with the smallest out degree is an attempt to maintain a low number of *sending events* which are incurred when executed on the target architecture. Let the current out degree of a cluster c' be denoted by $odg(c')$. The minimal degree among the candidate clusters is $d = \min_{\forall c' \in \Delta'(c)} odg(c')$. The subset of candidate clusters with out degree greater that d is constructed:

$$D = \{c' \in \Delta'(c) \ : \ odg(c') \neq d\}$$

Then, the remaining candidate clusters are the ones with minimal out degree, i.e.

$$\Delta'(c) = \Delta'(c) - D$$

Note that clusters which are still *useless* and are candidates have degree d equal to zero, which is the minimum degree among the candidates. If the resulting set of candidate clusters is such that $\mid \Delta'(c) \mid = 1$, an immediate predecessor cluster $c' \in \Delta'(c)$ of c is found. Otherwise, the search continues.

5.4.4.6 Rule 6: Same interval

If $\Delta'(c)$ contains more than one candidate cluster, refinements should be carried out on this set. In the beginning, when constructing the set of candidate clusters $\Delta(c)$, Rule 1 selects the clusters which have their finish time by $s(c, p) - \gamma$.

Therefore, it might be the case that there exist clusters which finish at earlier intervals than the adjacent interval to c interval.

Let us evaluate the following situation in Figure 5.5 in which there exist two clusters c' and c^* in $\Delta'(c)$. The cluster c' finishes at time $s(c,p)-(4\gamma+2)$, that is, c' is executed at two *computational* intervals earlier than the interval at which the visited cluster c is executed, while c^* is executed at time $s(c,p)-(2\gamma+1)$. Both clusters c' and c^* have the same priority, thus, suppose that c' is selected as an immediate predecessor cluster of c and the superedge (c', c) is defined. As a result, $odg(c') = 1$. The next phase of the $superdag$ algorithm (the phase in which *non-useless* clusters scheduled at time $s(c,p)-(2\gamma+1)$ are visited) the two visited clusters are c^* and some other cluster, say, c''. Suppose that c'' is such that its only candidate cluster is c' (see this situation in Figure 5.5(a)). As a result, the out degree of c' becomes equal to 2.

On the other hand, if, during $phase_c$, the cluster c^* is selected as an immediate predecessor cluster of c, when performing $phase_{c''}$ and $phase_{c^*}$, the result out degree of c' would be 1 (as seen in Figure 5.5(b)).

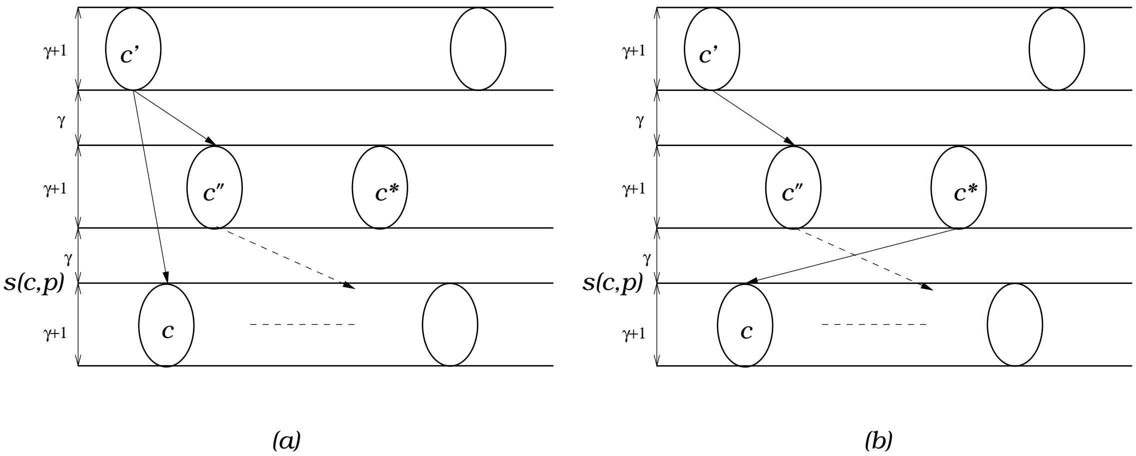


Figure 5.5: Breaking ties by using the latest candidate clusters. In (a) a non desirable situation is produced if the latest candidate cluster is not selected. The situation in (b) shows the better situation where a latest candidate is selected.

In order to avoid such unnecessary increase of out degree, the rule established to break ties at the current state of the search process, is to maintain in the set of candidate clusters $\Delta'(c)$ the ones that are executed as late as possible. Defining

$$t = \max_{c' \in \Delta'(c)} f(c', p'),$$

where the *virtual* processor p' is allocated to the cluster c' as specified in S, the subset of candidate clusters that should be eliminated from $\Delta'(c)$ is:

$$T = \{c' \in \Delta'(c) \ : \ f(c', p') \neq t\}$$

thus, the resulting set of candidate clusters is

$$\Delta'(c) = \Delta'(c) - T$$

The update of the set $\Delta'(c)$ by applying the above rule aims to give better possibilities for yet to be visited clusters, since in later phases, clusters will, in general, have less candidates than the ones visited in earlier phases. Again, if, as a result, $|\ \Delta'(c)\ | = 1$, an immediate predecessor cluster $c' \in \Delta'(c)$ of c is found. Otherwise, the last rule is applied, as we shall see next.

5.4.4.7 Rule 7: Owners with lower topological order

Let c' and c^* be clusters in $\Delta'(c)$. Hence, these candidates are scheduled in the same *computational* interval. Also, both contain the same number of immediate predecessors tasks in $\mathcal{I}'(c)$, that is,

$$x \ = |\ c' \cap \mathcal{I}'(c)\ | = |\ c^* \cap \mathcal{I}'(c)\ |$$

as a result of the previous rule application, and therefore, whichever candidate cluster is selected, the amount of data sent through the respective superedge is the same. Added to these two characteristics, both candidates have the same out degree d.

Let the tasks v_i, v_j and v_k be the owners of the clusters c, c' and c^*, respectively. Due to the topological order of visits $\mathcal{T}(v_i) \geq \mathcal{T}(v_j)$ and also, $\mathcal{T}(v_i) \geq \mathcal{T}(v_k)$. Suppose, w.l.o.g., that

$$\mathcal{T}(v_j) > \mathcal{T}(v_k) \tag{5.4}$$

The following outcomes are possible:

Case 1: *v_j-cluster is selected as an immediate predecessor cluster of c.* The shortest path from the source task in G (the original *DAG*) to the task v_j is $\mathcal{T}(v_j)$ long, which is the lower bound on the time elapsed from the source to the completion of v_j. Therefore, the time in which the processor allocated to cluster c can start executing the *receiving event* tasks followed by the tasks in c is $S_1 = \mathcal{T}(v_j) + x\tau + d\lambda_s$, considering the values for the sending overhead and delay on the communication.

Case 2: *v_k-cluster is selected as an immediate predecessor cluster of c.* Similarly, the time in which the processor allocated to cluster c can start execute the *receiving event* tasks followed by the tasks in c is $S_2 = \mathcal{T}(v_k) + x\tau + d\lambda_s$.

Therefore, due to Inequality 5.4, $S_2 < S_1$ holds.

The above discussion is the proof of the following Lemma:

Lemma 5.4.3 *Suppose that c is a visited cluster and that each candidate cluster contains the same number of tasks in $\mathcal{I}'(c)$. Furthermore, suppose that all of the candidates have the same current out degree d and also, the candidates are executed during the same interval in S. If the candidate cluster with smallest topological order is selected as an immediate predecessor cluster of c, then the schedule time of c is the earliest one.*

The idea behind this principle is to avoid the selection of candidate which have common tasks with the visited cluster c, i.e to avoid replication which increases the critical path of the resulting superdag, instead of minimising the resulting number of communication events.

5.4.5 Summarising the rules

The rules established in order to select reasonable and necessary cluster among the candidates as an immediate predecessor cluster of c, are evaluated in the order shown previously. **Rule 1** and **Rule 2** do not specify any superedge, but restrict the search space. The evaluation of each of the rules from 3 to 7 can provide an immediate predecessor cluster. Therefore, as soon as a **Rule** i selects an immediate predecessor cluster, the remaining rules do not need to be evaluated during the current iteration.

5.4.6 The algorithm

The *superdag* algorithm specified in Algorithm 6 determines the communication among the clusters, given the input *DAG* $G = (V, E)$ and the schedule S produced by the *replication* algorithm, which produces the set of clusters $V_{cluster}$. Algorithm 6 specifies the supertasks and the superedges, creating a *superdag* $G^* = (V^*, E^*)$.

A list, referred to as $Predecessors(c)$, is used to keep the superedges that are defined during $phase_c$. At the beginning of each phase of the $superdag$ algorithm, $Predecessors(c)$ is empty. At each time that a candidate $c' \in \Delta'(c)$ is selected, c' is included in $Predecessors(c)$. Immediately after, the set of immediate tasks $\mathcal{I}'(c)$ must be updated to $\mathcal{I}'(c) = \mathcal{I}'(c) - \{c' \cap \mathcal{I}'(c)\}$ since the sender of the data $\{c' \cap \mathcal{I}'(c)\}$ has already been specified.

The initialisation of the necessary structures is specified from lines 1 to 3. In line 1 of $superdag$, the topological order of each vertex $v_i \in V$ is defined.

In line 2, the first clusters to be visited are listed in C. Using the topological order of their owners ensures that the order of the visits is carried out from the sink clusters to the sources. When the sources are visited, the $superdag$ procedure reaches its end, since no more clusters are included in C. In line 3, $useless(c)$ is properly set for all c-owner $\in V$, indicating whether the cluster c is *useless* or not. Line 4 specifies the loop which characterises the phases of the algorithm. For each cluster c visited, the corresponding set $\mathcal{I}(c)$ is constructed in line 7. Then, the structures $\mathcal{I}'(c)$ and $Predecessors(c)$ are initialised in line 8. After that, the set of all possible candidate clusters $\Delta(c)$ is established in line 9, corresponding to the implementation of the first rule. Line 10 represents the beginning of each iteration and also, it is where the remaining rules are implemented. The second rule is carried out between lines 11 and 14, where the restrict set of candidate clusters $\Delta'(c)$ is constructed. For this, in lines 11 and 12 the tasks in $\mathcal{I}'(c)$ which are less replicated are identified. Then, Rule 3 is executed an attempt to define a superedge. It is verified if there are tasks in $\mathcal{I}'(c)$ which are replicated only once in the candidates. For all the candidates containing such tasks, an immediate predecessor is defined by executing lines 16 to 20. If Rule 3 does not apply, the set of candidates is decreased by keeping the clusters with the most tasks in $\mathcal{I}'(c)$, corresponding to the Rule 4. Line 21 calculates $ntasks(c', c)$ for each candidate and $\Delta'(c)$ is decreased as shown from lines 22 to 24. If there exists only one resulting candidate cluster, an immediate predecessor is defined as indicated by lines 28 and 29. On the other hand, if there are still two or more candidate clusters in $\Delta'(c)$, the next rules are carried out. These rules are specified in the procedure $break\_ties$, executed in line 26 of the Algorithm 6.

Algorithm: 6 $superdag()$ {

1 define $\mathcal{T}(v_i)$, $\forall v_i \in V$; $w((c',c)) := 0$, $\forall (c',c) \in V_{cluster} \times V_{cluster}$;

2 $C := \langle c_1, \ldots c_k \rangle$: c_j-owner $\in V \wedge succ(c_j$-owner $) = \emptyset$, $j = 1, \ldots, k \wedge \mathcal{T}(c_1$-owner $) \geq \ldots \geq \mathcal{T}(c_k$-owner $)$;

3 $\forall v_i \in V,\ useless(v_i) := \begin{cases} FALSE & \text{if } succ(v_i) = \emptyset; \\ TRUE & \text{otherwise} \end{cases}$

4 while $(C \neq \emptyset)${ /* *Phases* */

5 c is the first element of C;

6 $C := C - \{c\}$;

7 $\mathcal{I}(c) := \{u_k \in V\ :\ (u_k, v_{i_j}) \in E \wedge u_k \notin c \wedge v_{i_j} \in c\}$;

8 $\mathcal{I}'(c) := \mathcal{I}(c)$; $Predecessors(c) := \emptyset$;

9 $\Delta(c) := \{c' \in V_{cluster}\ :\ s(c',p') \leq s(c,p) - \gamma \wedge \exists u_j \in c'$ such that $u_j \in \mathcal{I}(c)\}$;

while $(\mathcal{I}'(c) \neq \emptyset)$ { /* *Iteration* */
$\forall u_i \in \mathcal{I}'(c)$
$Ncluster(u_i) := \sum_{c' \in \Delta'(c)} \mid \{u_i\} \cap c' \mid;$
$y := \min_{\forall u_i \in \mathcal{I}'(c)} Ncluster(u_i);$
$\Delta'(c) := \{c' \in \Delta(c) \ : \ u_i \in c' \ \wedge \ Ncluster(u_i) = y\};$
if $(y = 1)$ {
while $((\Delta'(c) \neq \emptyset) \ \wedge \ (\mathcal{I}'(c) \neq \emptyset))$ {
$c' := c'' \in \Delta'(c);$
$\Delta'(c) := \Delta'(c) - \{c'\};$
$Predecessors(c) = Predecessors(c) \cup \{c'\};$
$\mathcal{I}'(c) = \mathcal{I}'(c) - \{c' \cap \mathcal{I}'(c)\};$
}
}else {
$\forall c' \in \Delta'(c), \ ntasks(c') := \mid c' \cap \mathcal{I}'(c) \mid$
$x := \max_{c' \in \Delta'(c)} \{ntasks(c')\};$
$X := \{c' \in \Delta'(c) \mid ntasks(c') \neq x \};$
$\Delta'(c) := \Delta'(c) - X;$
if $(\mid \Delta'(c) \mid > \ 1)$
$c' := break\_ties(\Delta'(c));$
else
$c' := c'' \in \Delta'(c);$
$Predecessors(c) = Predecessors(c) \cup \{c'\};$
$\mathcal{I}'(c) = \mathcal{I}'(c) - \{c' \cap \mathcal{I}'(c)\};$
}
}
$add\_edges(Predecessors(c), \mathcal{I}(c));$
}
}

The procedure $break\_ties$ implements Rule 5, as specified in lines 1 to 3 of Procedure 7. If there is still more than one candidate in $\Delta'(c)$, Rule 6 is applied. For this, the latest time amongst the times of the candidates is calculated in line 5 of $break\_ties$, and the set $\Delta'(c)$ is reduced in lines 6 and 7. If there are still two or more candidates, the last rule is performed in lines 9 to 11, where one of the candidates, whose owner has the smallest topological order, is selected.

Procedure: 7 $break\_ties(\Delta'(c))$ {
1 $d := \min_{c' \in \Delta'(c)} \ \{odg(c')\};$

$D := \{c' \in \Delta'(c) \ : \ odg(c') \neq d\};$

$\Delta'(c) = \Delta'(c) - D;$

if $(\mid \Delta'(c) \mid > \ 1)\{$

$\quad t := \max_{c' \in \Delta'(c)} f(c', p');$

$\quad T := \{c' \in \Delta'(c) \ : \ f(c', p') \neq t\};$

$\quad \Delta'(c) := \Delta'(c) - T;$

$\}$else $\{$

$\quad c' := c'' \in \Delta'(c);$

$\}$

if $(\mid \Delta'(c) \mid > \ 1)\{$

$\quad l := \min_{c' \in \Delta'(c)} \{\mathcal{T}(c'\text{-owner })\};$

$\quad$ Let c' be an arbitrary element of $\Delta'(c)$ such that $\mathcal{T}(c'\text{-owner }) = l;$

$\}$

return $c';$

$\}$

The add_edges procedure establishes the superedges between the selected immediate predecessor clusters and the visited cluster c. The procedure is called at the end of $phase_c$ (line 30 of the *superdag* algorithm), when $\mathcal{I}'(c) = \emptyset$. The list $Predecessors(c)$ contains the necessary and sufficient immediate predecessors of the visited cluster c. In procedure add_edges, for each $c' \in Predecessors(c)$, a superedge is defined. However, an ordering on the clusters (supertasks) in $Predecessors(c)$ is introduced. Firstly, the set $\mathcal{I}(c)$ (Definition 5.4.1, page 105) is reassigned the set $\mathcal{I}'(c)$, as seen in line 1 of the add_edges procedure. After that, the set $Predecessors(c)$ is sorted in the increasing order of the amount of data sent by each of cluster $c' \in Predecessors(c)$, as given in the ordered list $\mathcal{L}$ associated with $Predecessors(c)$:

$$\begin{aligned} \mathcal{L} = \ & \langle c_{i_1}, \ldots c_{i_l} \rangle \ : \ c_{i_j} \in Predecessors(c), \ j = 1, \ldots, l \ \wedge \\ & ntasks(c_{i_1}, c) \ \leq \ldots \leq \ ntasks(c_{i_l}, c) \end{aligned}$$

where $l = \mid Predecessors(c) \mid$ (line 2). While both $\mathcal{L}$ and the set $\mathcal{I}'(c)$ are not empty, lines 5 to 11 define the superedge (c', c), where c' is the first cluster in the ordered list $\mathcal{L}$ (line 4). The ordering of the specification given by $\mathcal{L}$ is an attempt to avoid the definition of superedges with non-uniform weights and does not increase the current in degree and out degree of cluster c. In line 5 the out degree $odg(c')$ of the supertask c' is incremented and then, c' becomes *non-useless* since it is selected to be an immediate predecessor cluster (it is a supertask now), and consequently, it is only incorporated into the list $\mathcal{C}$ if it has not been visited yet (lines 6 to 8). The weight of the superedge (c', c) is equal to $\mid \mathcal{I}'(c) \cap c' \mid$, which is the necessary amount of data transmitted by c' to c (line 9). As there is no need for this data to be sent again by clusters other than c', the tasks in $\mathcal{I}'(c) \cap c'$ are removed from $\mathcal{I}'(c)$ (lines 10 and 11).

Procedure: 8 $add\_edge(Predecessors(c), \mathcal{I}(c))$ {

1 $\quad \mathcal{I}'(c) := \mathcal{I}(c);$

2 $\quad \mathcal{L} = \langle c_{i_1}, \ldots c_{i_l} \rangle \; : \; c_{i_j} \in Predecessors(c), \; j = 1, \ldots, l \, \wedge$ $ntasks(c_{i_1}, c) \; \leq \ldots \leq \; ntasks(c_{i_l}, c);$

3 $\quad$ while $((\mathcal{L} \neq \emptyset) \; \wedge \; (\mathcal{I}'(c) \neq \emptyset))$ {

4 $\qquad c'$ is the first cluster in $\mathcal{L}$;

5 $\qquad odg(c') := odg(c') + 1;$

6 $\qquad$ if $useless(c')${

7 $\qquad\quad$ insert c' in C according to its topological order;

$\qquad$ }

8 $\qquad useless(c') \; := \; FALSE;$

9 $\qquad w((c', c)) \; := \mid \mathcal{I}'(c) \cap c' \mid;$

10 $\qquad R \; := \; \mathcal{I}'(c) \cap c';$

11 $\qquad \mathcal{I}'(c) \; := \; \mathcal{I}'(c) - R;$

$\quad$ }

}

5.5 The resulting superdag

After the superedges have been defined, it is easy to know which are the supertasks of the resulting superdag $G^* = (V^*, E^*)$. $V_{cluster}$ is the input set of all of the clusters. The set of supertasks is defined as:

$$V^* = \{c \in V_{cluster} \mid useless(c) = FALSE\}$$

The set of superedges is:

$$E^* = \{(c', c) \in V_{cluster} \times V_{cluster} \mid w((c', c)) \neq 0\}$$

The superdag G^* is then ready to be used as the input *DAG* to the remaining stages of the *multi-stage approach*, in which, other parameters of the target architecture are incorporated.

We show now a characteristic of the superdag produced which will be necessary for futures stages of MSA, as we will see. The following Lemma establishes that at each band defined by the *replication* algorithm, there will remain a supertasks in the final superdag.

Lemma 5.5.1 *Let $G^* = (V^*, E^*)$ be the superdag that is produced by the first stage of the multi-stage approach and let S be the schedule produced by the replication algorithm. There exists at least one supertask $c \in V^*$ scheduled during each computational interval β defined in S.*

Proof:

From the $superdag$ algorithm, at least one of the sink supertasks in V^* must be scheduled during band 0.

Let $c \in V^*$ be scheduled during interval β. The supertask c represents the cluster of tasks $c = \langle v, u_1, u_2, \ldots, u_\gamma \rangle$ in which the task $v \in V$ is its owner. A list of all of the ancestors of v, given in their decreasing order of e_l-values, is represented by $elist_l(v) = \langle u_1, u_2, \ldots, u_\gamma, u_{\gamma+1}, \ldots, u_t \rangle$, where t is the total number of ancestors of v. The e_l-value of the task $u_{\gamma+1}$ is used to calculate $e_l(v)$ and therefore, the v-cluster is scheduled to start execution at time $2e_l(v) - \gamma$ in the v-cluster, since task v is scheduled at time $2e(v)$. Consequently, $u_{\gamma+1} \in \mathcal{I}(c)$, so the data from this task must be sent to the cluster c. Corollary 5.3.1 establishes that all of the copies of the task $u_{\gamma+1}$ are scheduled within the interval of time:

$$[2e_l(u_i) - \gamma, 2e_l(u_i) + 2\gamma + 2]$$

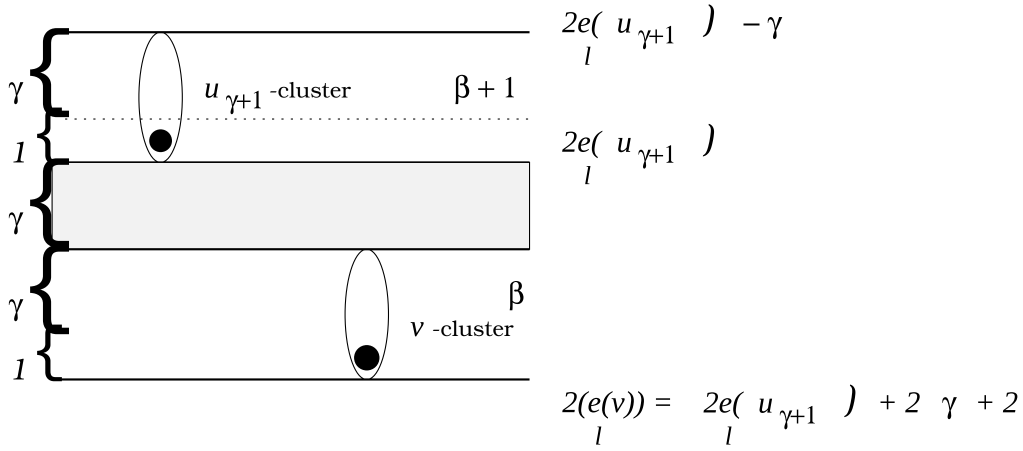


Figure 5.6: Interval β contains the v-cluster, which needs information from a cluster scheduled during interval $\beta + 1$

The diagram in Figure 5.6 shows that the clusters in which copies of $u_{\gamma+1}$ are scheduled in time to send their data to the cluster c are within band $\beta + 1$. Therefore, there must exist a cluster $c' \in V^*$, that is scheduled to execute during interval $\beta + 1$ in which $u_{\gamma+1} \in c'$.

$\square$

5.5.1 A Complexity analysis of the *superdag* algorithm

The execution of the *superdag* algorithm begins with the construction of the list C which contains the clusters in $V_{cluster}$ to be visited. Initially, only those clusters containing the sink tasks are included in C. During the execution of the *superdag* algorithm, a cluster is included in C after it is selected as an immediate predecessor supertask. In the worst case, all of the clusters in $V_{cluster}$ are visited. Therefore, line 4 is executed $\mathcal{O}(n)$ times, where $n = \mid V \mid$.

The number of iterations in each phase is dictated by the size of $\mathcal{I}'(c)$, initialised as $\mathcal{I}(c)$, which can have at most $n-1$ tasks (i.e. when a cluster needs to receive data items from all of the tasks except its owner). Therefore, the number of iterations is $\mathcal{O}(n)$.

The loop starting at line 16, which implements the *Unique Choice* Rule, is also no more than $\mathcal{O}(n)$ since there can be at most n-1 (actually, $\mid \mathcal{I}(c) \mid$) candidates in $\Delta'(c)$ each sending a unique data item from $\mathcal{I}(c)$.

The remaining rules are implemented based on the set of candidates $\Delta'(c)$. This set can have at most $n-1$ clusters, which occurs when a sink cluster depends on the other remaining clusters (given that the *replication* algorithm produces n clusters). The algorithm that selects the minimum or the maximum value among n values takes $\mathcal{O}(n)$ time [30]. Therefore, each one of the remaining rules can be implemented in $\mathcal{O}(n)$ time. Concluding, the *superdag* algorithm runs in $\mathcal{O}(n^3)$.

The worst case complexity, $\mathcal{O}(n^3)$, is related to execution time. In practice, the execution time may not be order of n^3. As an example, when there is only a single copy of each task, the loop at line 10 is executed only once. Considering the worst case for the loop at line 16, the Algorithm has complexity $\mathcal{O}(n^2)$. For some *DAGs*, replication of tasks is not necessary when constructing the superdag (e.g. binary trees). On the other hand, when replication is necessary for the minimisation of communication costs, more options are available as immediate predecessor supertasks and this therefore increases the complexity.

5.6 On the number of processors

Jung *et al.* in [70] discussed the value of task replication showing that there may exist optimal schedules without task replication that are a multiplicative factor of the true optimal schedules with replication, considering the *delay* model of Papadimitriou and Yannakakis. As in the Papadimitriou and Yannakakis' heuristic, the *replication* algorithm implemented in the first stage of

MSA allocates each task to a different *virtual* processor, and therefore, there is no need for more processors than the number of tasks [123]. Therefore, for large *DAG*s, a large number of processors is necessary. In this case, supertasks in the correspondent superdag may be merged onto a single physical processor if the number of physical processors is much smaller than the number of supertasks.

Although the *replication principal* is used in the first stage of MSA without considering the *delay* model, but used to cluster tasks so that messages are bundled, we come back to the *delay* model of Papadimitriou and Yannakakis and evaluate the possibility of incorporating another architectural characteristic into their model. We now incorporate a notion of locality to the *delay* model and through the schedule of a given example, we establish the effects that this incorporation can produce on the optimal makespan.

5.6.1 The *locality* model

In order to evaluate locality in the *delay* model, the *locality* model is defined as in the manner specified in [125]. In many real architectures, the processors are connected under a certain topology so that, each processor is connected to a restrict number of other processors. An undirected graph that represents this kind of network is not completely connected one, but all of the nodes have the same degree d. In the *locality* model, the following delays may be incurred when processors communicate:

τ_a **:** delay which is incurred when adjacent processors communicate;

τ_n**:** delay which is incurred when any non-adjacent processors communicate;

As usual, if communicating tasks are allocated to the same processor, the delay on the communication is considered negligible.

5.6.2 An example

In order to show how locality can affect the optimal schedule which incorporates task replication, the example constructed in [125] is analysed. The example is the *layered DAG* $G = (V, E)$ shown in Figure 5.7.

The *DAG* G is constructed given an index n, which specifies its number of levels. The set of vertices V and the set of edges E can be divided into the following disjoint sets:

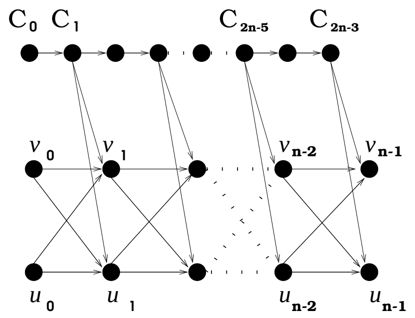


Figure 5.7: A layered *DAG* $G = (V, E)$, indexed by n, the number of layers

- those vertices that forms a chain and their edges:

$$C = \{c_i \ : \ i = 0, \ldots, 2n-3\}$$
$$E_c = \{(c_{i-1}, c_i) \ : \ i = 1, \ldots, 2n-3\}$$

- those vertices that forms a "ladder" and their edges:

$$\begin{aligned}
V_v &= \{v_i, \ : \ i = 0, \ldots, n-1\} \\
V_u &= \{u_i, \ : \ i = 0, \ldots, n-1\} \\
E_v &= \{(v_{i-1}, v_i) \ : \ i = 1, \ldots, n-1\} \\
E_u &= \{(u_{i-1}, u_i) \ : \ i = 1, \ldots, n-1\} \\
E_{vu} &= \{(u_{i-1}, v_i), (v_{i-1}, u_i) \ : \ i = 1, \ldots, n-1\}
\end{aligned}$$

- the edges that join the chain to the ladder:

$$E' = \{(c_{2i-1}, v_i), (c_{2i-1}, u_i), \ : \ i = 1, \ldots, n-1\}$$

Finally,

$$\begin{aligned}
V &= C \cup V_v \cup V_u \\
E &= E_c \cup E_v \cup E_u \cup E_{vu} \cup E'
\end{aligned}$$

In the analysis, the values of the architectural parameters are: $d = 3$, $\tau_a = 1$, $\tau_n = 2$. We wish to find an optimal schedule for the *DAG G* considering the *locality* model.

5.6.3 The schedule

Firstly, the earliest start time of the tasks in the optimal schedule is established. Each task c_i in the chain C cannot start before time step i, since c_i cannot start before all of its predecessors are executed. The latest immediate predecessors of each task $v_i \in V_v$ (and $u_i \in V_u$) is the task c_{2i-1}. Therefore, v_i (and u_i) cannot start before time $2i$ in the optimal schedule.

Now, we construct an optimal allocation for G, considering the index n, $d = 3$, $\tau_a = 1$ and $\tau_n = 2$. Initiating the construction by evaluating the optimal start time of task $v_{n-1} \in V_v$, this task is scheduled on a certain processor together with all the chain task predecessors $c_0, \dots c_{2n-3}$. Task v_{n-1} needs to receive data from its immediate predecessors v_{n-2} and u_{n-2} which, by construction, are allocated to distinct processors, each one together with a copy of the chain $c_0, \dots c_{2n-5}$. In the same manner, the processor allocated to v_{n-2} receives data from two other processors allocated to v_{n-3} and u_{n-3}. Also, the processor allocated to u_{n-2} needs to receive data from the processors allocated to v_{n-3} and u_{n-3}. However, since the processors have degree $d = 3$ the tasks v_{n-3} and u_{n-3} are replicated onto another two processors, respectively (together with their chain). Inductively, v_i at level i is allocated together with its chain onto a distinct processor, receiving data from v_{i-1} and u_{i-1}, which are allocated to distinct processors with their respective chains. The tasks v_i and u_i, at level i, are replicated with their chains onto 2^{n-i} processors. The task v_i is allocated to 2^{n-i-1} processors, identified as p^j_{n-i}, where $j = 0, 2, 4, \dots, 2^{n-i}$ and task u_i is replicated onto processors p^j_{n-i}, where $j = 1, 3, \dots, 2^{n-i} - 1$.

Obviously, task v_0 is scheduled at time 0 since task v_{i-1} is scheduled at time $2(i-1)$ on processor $p^j_{n-(i-1)}$, then task v_i is scheduled on processor p^j_{n-i} which has been busy since time step zero executing the tasks $c_0, \dots c_{2i-1}$. Note that if v_{i-1} is scheduled on p^j_{n-i}, the chain tasks are not executed in optimal time.

In order to execute task v_i at time $2i$, processor p^j_{n-i} must receive data from v_i's immediate predecessor by this time. These tasks are executed by time $2(i-1)$, a delay of τ_a time steps is incurred between the processors allocated to v_{i-1} and u_{i-1} and the processor p^j_{n-i}. Therefore, there must exist two processors adjacent to p^j_{n-i}. Consequently, the makespan of the above allocation of G is $2n-1$ with Figure 5.8 showing the optimal allocation. The dependency relation between the processors is established. The dependency relation is isomorphic to two disjoint binary trees, each one with $n-1$ layers (the only copy of task v_{n-1} is allocated to one sink processor and task u_{n-1} to the other). Therefore, the optimal schedule of G under the *locality* model requires $2^{n+1} - 2$ physical processors.

The above discussion is the proof for the following theorems, which are also formalised in [125]:

Theorem 5.1 *Let G be the layered DAG with index n and let $d = 3$, $\tau_a = 1$ and $\tau_n = 2$. Then there exists a schedule for G with makespan $2n-1$.*

Theorem 5.2 *Any valid schedule for the layered DAG G with index n that has a makespan no greater than $2(n-1)$ requires at least $2^{n+1} - 2$ processors.*

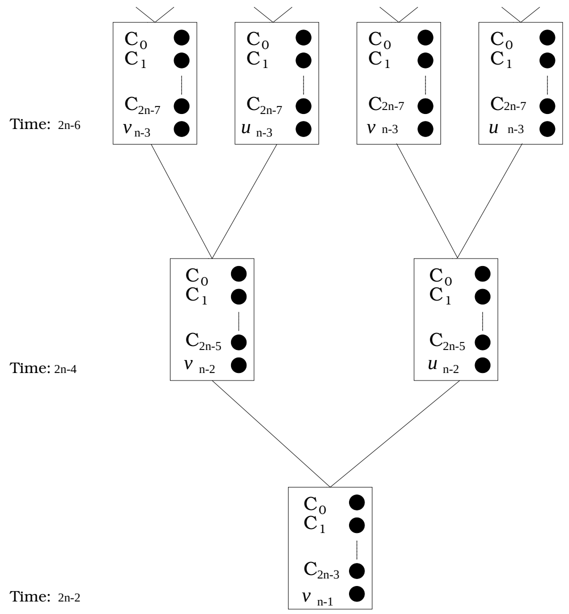


Figure 5.8: The layered *DAG* allocation, with index n, $d = 3$, $\tau_a = 1$, $\tau_n = 2$

5.7 Concluding remarks

It is difficult to consider many architectural characteristics at once in a one stage heuristic. The *replication principle* of Papadimitriou and Yannakakis gives good schedules considering their model, but if locality is incorporated, the optimal makespan can only be achieved under possibly unacceptable physical requirements. The ETF algorithm is devised under a model that includes locality but as it has been already shown, it can produce schedules that are not very "good".

These conclusions establish the validity of devising several stages for a schedule strategy when many architectural parameters are considered. The *replication principal* of Papadimitriou and Yannakakis is very promising when attempting to cluster tasks, not only with the objective of minimising delay on the communication but also, to decrease the number of *communication events* which are incurred.

Concluding the first stage of the *multi-stage approach*, tasks of a *DAG* $G = (V, E)$ are clustered together, given the *clustering factor* γ, in order to decrease the cost associated with the communication events which may be incurred. As a result, a superdag $G^* = (V^*, E^*)$ is provided through the specification of communication among resulting supertasks. Also, a schedule S of the superdag $G^* = (V^*, E^*)$ is defined in a very simplistic way. Note that the values of the parameters of the architecture specified in the CLAUD model are not considered, but their essence are taken into account. The main objective of the first stage is to cluster tasks and to define a new relationship in order to reduce the communication costs, rather then to define a definitive schedule for the input application. Next, the second stage of our strategy is proposed, where these parameters are properly incorporated and the final schedule considering the CLAUD model is produced.

Chapter 6

Remapping Supertasks onto a given set of Processors

6.1 Introduction

For the input *DAG* $G = (V, E)$ and the *clustering factor* γ, the first stage of the *multi-stage approach* clustered the tasks of the *DAG* G, taking into account the precedence relationship between them to produce a superdag $G^* = (V^*, E^*)$ by specifying the communication among resulting supertasks. A schedule S of the superdag $G^* = (V^*, E^*)$ which includes information about the clustered tasks, is provided, where each cluster is allocated to a distinct *virtual* processor.

Using the tuples of the schedule S, it is possible to identify sets of parallel supertasks in V^*. Each set is only executed after all of the non-local data is made available to each of its supertasks. When the execution of the parallel supertasks has finished, data can be sent to their respective immediate successors. The regularity that results from this definition of the schedule times in S provides an easier way to analyse the set of parallel supertasks.

The main objective of the first stage is to cluster tasks into supertasks and to define their relationship in order to reduce the communication costs, rather than to specify a definitive schedule for the input application. Given that each supertask is allocated to a distinct *virtual* processor and the out degree and in degree of each supertask are not necessarily one, the actual schedule S does not capture the characteristics of the target model.

However, as the determination of a reasonable schedule under the CLAUD model is the objective, further transformations and refinements are necessary. The second stage of the *multi-stage approach* incorporates the important architectural characteristics defined in the CLAUD model such as the overhead λ_s and λ_r; the delay τ; and the number m of physical processors available in the network. The function $\varphi : \mathcal{P} \times \mathcal{P} \longrightarrow Z_0^+$ (see page 44) is not used in this stage, because the current topology assumed is the fully connected network.

Given the number of processors m, it may be necessary to reallocate a certain number of supertasks onto a single physical processor. It is very important for the effectiveness of the scheduling process that the reallocation of supertasks does not lead to an increase in the number of communication events. Therefore, the current stage must follow similar guidelines to the ones used during the previous stage of MSA: messages are bundled in such a way that the degree of resulting supertasks is maintained as low as possible and consequently, a relatively small number of communication events is incurred when the new superdag is executed in parallel.

At the same time that mechanisms are implemented to achieve the above objective, the degree of parallelism is also exploited by equally reallocating each set of parallel supertasks onto m processors. This method provides a higher degree of parallelism than in the one in which G^* is "transformed" into a new superdag with m new supertasks.

The well-defined locations of the sets of parallel supertasks along the interval $[0, \mathcal{M}(S)]$ are of crucial importance for the implementation of transformations (the remapping and merging of supertasks) at the current stage. The parallel supertasks within each set have the same execution cost and are precisely scheduled within an interval $\gamma + 1$ long. Each group of supertasks reallocated to a single processor is "merged" and transformed into a new supertask. Due to the existence of sets of parallel supertasks, Brent's Principle is applied in order to derive the number of supertasks reallocated to each of the m processors. The Principle specifies that, during a given step, say, i, a certain number of operations can be done in parallel. In our context, the "step" corresponds to an interval, and the "operation" to a supertask.

Brent's Principle does not specify the processors to which parallel supertasks are reallocated, since intuitively, this specification may not be necessary. However, due to the precedence relationship among supertasks scheduled on adjacent intervals and the characteristics of the target model, the selection of processors is essential for the effectiveness of the schedule approach.

In the literature, there exist heuristics, like the clustering algorithms, which initially suppose an unbounded number of processors (or at least as many processors as tasks of the input *DAG*) and the tasks are scheduled to *virtual* processors. After that, if there are more *virtual* processors than physical ones, a "merge" mechanism is necessary. Other heuristics, like list scheduling heuristics, accept as input, the number of physical processors and produce a schedule for the input *DAG* considering the available physical processors. There are cases that the schedule does not necessary use all of the processors. Also, due to the near-optimality of the heuristics, it could be the case that if a smaller number of processors where given to the heuristic, a schedule with a smaller makespan would be achieved. Heuristics have this behaviour mainly because their objective is to find a schedule with minimal makespan rather than to solve the *max-min* problem: maximise the degree of parallelism while minimising the communication costs.

6.1.1 Brent's Principle

The basic idea of Brent's scheduling Principle [20] is to exploit the degree of parallelism of an application. Suppose that in a give step i of a parallel computation compounded of a total of $\mathcal{T}$ steps, x_i operations can be done in parallel, without any communication being required, so that $\sum_{i=0}^{\mathcal{T}} x_i = \mathcal{M}$. Suppose that there exists a total of m available processors. It is possible to simulate the computation effectively by evaluating the ith step in time

$$\left\lceil \frac{\lfloor x_i \rfloor}{m} \right\rceil \leq \frac{x_i}{m} + 1$$

Hence, the total parallel time can be simulated in no more than

$$\frac{\mathcal{M}}{m} + \mathcal{T}$$

In his definition, Brent assures that the allocation of tasks is not a problem, what it is not the case in out context. Also, communication costs is not an issue considered in [20]. Despite the simplicity of the assumptions stated by Brent, the Principle has been used to design efficient parallel algorithms [72, 126].

6.1.2 Remapping using Brent's Principle

Thurimella and Yesha [126] based their work on the *delay* model of Papadimitriou and Yannakakis. They developed a polynomial time *remap* heuristic that transforms a schedule S of a precedence graph with n tasks to a schedule S_m onto m physical processors, considering that $n > m$. An application is represented by a *DAG* $G = (V, E)$ where each task in V has unit execution cost and each edge has a unit weight. The unique parameter of the architecture taken into account is the constant communication delay τ. The schedule S considered as input by Thurimella and Yesha, results from Papadimitriou and Yannakakis' heuristic and makes use of a set of n processors, denoted by $\mathcal{Q} = \{q_1, q_2, \ldots, q_n\}$. Thurimella and Yesha's heuristic (TY) applies Brent's Principle in order to reallocate parallel tasks onto the given m processors, which are enumerated as $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$. In order to identify these parallel tasks, the schedule S is partitioned onto stripes and the set of tuples in S is reorganised onto a smaller number of processors.

Let $\mathcal{M}(S)$ be the makespan of S. The time axis, which varies from 0 to $\mathcal{M}(S)$, is partitioned into stripes τ units height. The tuples correspondent to the schedule of the tasks executed during stripe, say, j, is $S_j \subseteq S$. Let $tasks(p_i)$ denote the set of tasks reallocated to $p_i \in \mathcal{P}$. TY is compounded two main steps, which are described as follows:

1. The tuples in S_j are evenly divided between the m processors in $\mathcal{P}$, through the application of Brent's Principle. All tasks allocated to a processor q_k at stripe j are assigned to a processor p_w and incorporated to $tasks(p_w)$. If $\mid tasks(p_w) \mid > \lceil \mid S_j \mid /m \rceil$ then, another processor $p_{w+1} \in \mathcal{P}$ is selected, so that the remaining tasks in S_j, not yet reassigned, are reallocated. This simple algorithm is repeated until all tasks in S_j executed at each stripe j are remapped;

2. Each stripe is τ units wide. Thus, for any processor q_k in the input schedule S, the maximum number of tasks executed at stripe j is τ. Then, after the *processor assignment* execution, no more then $\lceil \frac{|S_j|}{m} \rceil + \tau - 1$ is assigned to a processor p_w, since a set of tasks originally assigned to q_k are not be split in S_m. Between any two stripes an idle time of τ units time steps, to allow communication between adjacent tasks reallocated to distinct processors to be carried out. Thus, a task v_i of S_j should be scheduled in S'_j at time step given by:

$$t' = j\tau + rank_j(v_i) + \sum_{k=0}^{j-1} (\lceil \mid S_k \mid /m \rceil + \tau - 1)$$

   where $rank_j(v_i)$ is the order of execution of task v_i onto the new processor, say, p_k at stripe j.

In the definition of the algorithm, it is clear to see that there is always an idle time of τ time steps between any two stripes of tasks specified in the new schedule S_m. On the attempt to apply such remapping technique in models which consider the startup overhead, like in the CLAUD model, this idle period of time could be interpreted as the sending overhead λ_s.

However, some drawbacks can be pointed out in the remapping scheme of Thurimella and Yesha when considering the CLAUD model. Firstly, there is no specification on how to handle copies of the same task reallocated to the same processor, even when these copies are scheduled on different stripes. Secondly, there is no concern on the dependencies among the tasks. It must be borne in mind that *communication events* are incurred when adjacent tasks are reallocated onto different processors. Also, there is no specification of bundling of messages and therefore, the algorithm TY as it is would not produce a reasonable schedule for the target model.

We consider that, when tasks are reallocated to a processor p_k at stripe j, the sending of messages is carried out only after all the reallocated tasks are executed. In this manner, suppose the situation illustrated in Figure 6.1. On

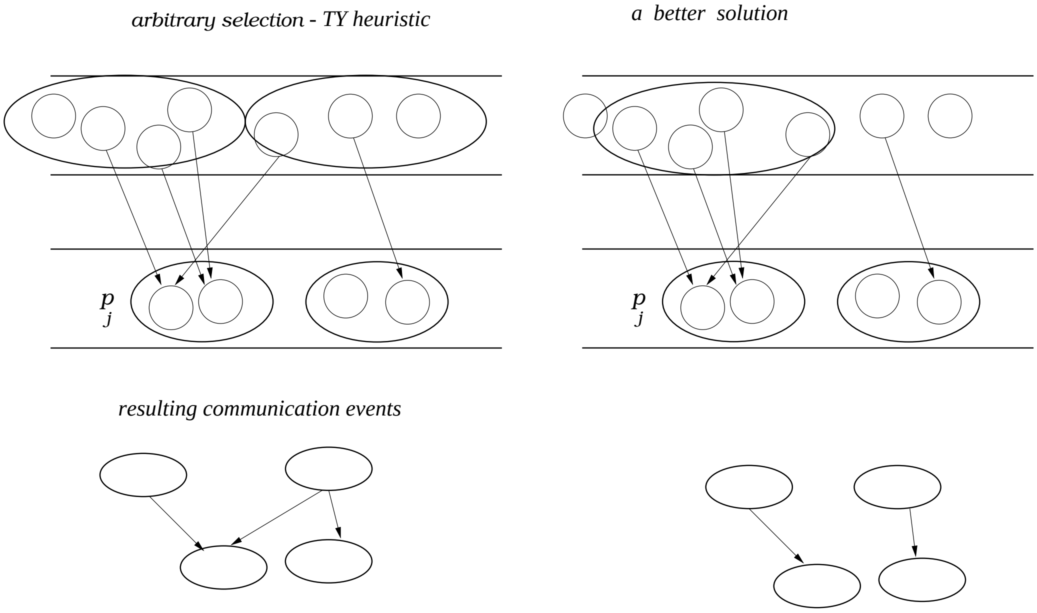


Figure 6.1: On the number of *communication events* when applying TY technique.

the top left side, tasks are selected with no concern on the precedence relation among them, i.e. the TY heuristic is applied in its essence. On the down left side, the reallocated tasks are clustered and the *communication events* are represented by the edges produced. In one of the clusters, two *sending events* are represented, but only one affects the makespan, since the second is incurred between clusters reallocated to the same processor. However, a better schedule for our target model can be provided by analysing the precedence relationship between tasks. On the top right side of Figure 6.1, a better selection is carried out and the respective *communication events* are represented on the bottom right side. As a result, a proper schedule for the target model is produced, where no *communication events* between different processor are incurred.

6.2 The phases of the *remapping* algorithm

The second stage of the *multi-stage approach* is comprised of the reallocation of parallel supertasks onto m processors using mechanisms implemented in a proposed algorithm referred as the *remapping* algorithm. Two phases are identified in the *remapping* algorithm:

the *backwards phase*: the supertasks scheduled within the same interval are visited and reallocated to a fully connected network with m processors. During this phase, although the values of the architectural parameters λ_s, λ_r and τ are not taken into account, the essence of these parameters are considered. The main mechanism is based the reallocation of supertasks in such a way that the number of communication events may not increase. Briefly, the mechanism is described as follows: with the knowledge of the precedence relationship among supertasks, when reallocating a group of supertasks to the same processor, priority is given to a group in which the supertasks have common immediate successors, given that these immediate successors have already been remapped.

Given the schedule S, the supertasks in G^* are visited in the same order as the one defined in the $superdag$ algorithm (in the first stage of MSA): from the latest interval, during which the sink supertasks are scheduled in S, up to the interval during which the sources are scheduled. Although the final new schedule is not effectively defined during this phase, the *allocation* of the supertasks and the definition of a new superdag $G_s = (V_s, E_s)$ is produced.

the *updating phase*: during this second phase of the *remapping* algorithm, the new schedule S_s is precisely specified, given

- the new allocation provided by the *backwards* phase;
- the new superdag $G_s = (V_s, E_s)$;
- the weight associated with the new superedges;
- the execution costs of the new supertasks;
- the overheads λ_s and λ_r;
- and the delay τ.

In summary, given a superdag $G^* = (V^*, E^*)$ and its schedule S, both provided by the first stage of MSA, the second stage, involving the *remapping* algorithm, transforms the superdag G^* into a new superdag $G_s = (V_s, E_s)$, providing a schedule S_s of G_s onto m fully connected processors, taking into account the parameters λ_s, λ_r and τ, as defined in the CLAUD model.

6.3 The sets of parallel supertasks

Let S be the schedule of the input superdag $G^* = (V^*, E^*)$ and $\mathcal{M}(S)$ its makespan. The set $\mathcal{P} = \{p_0, p_1, \ldots, p_m\}$ is the set of m physical available processors. A *computational* interval or *band* during which parallel supertasks are scheduled in S is also denoted as β. The band during which the sink supertasks are scheduled, is the band 0, enumeration which increases up to band $\mathcal{B} - 1$, where the source supertasks are scheduled. Given S and γ, the value for $\mathcal{B}$ (as already derived in Section 5.2) is:

$$\mathcal{B} = \left\lceil \frac{\mathcal{M}(S) - (2\gamma + 1)}{2\gamma + 1} \right\rceil$$

The start time of each band β, denoted by $start(\beta)$, corresponds to start time of the parallel supertasks scheduled during β in S, i.e.

$$start(\beta) = 2((\mathcal{B} - 1 - \beta) * (\gamma + 1) + \gamma) - \gamma$$

The list of parallel supertasks scheduled during each band β, $0 \leq \beta \leq \mathcal{B} - 1$, denoted by $\mathcal{C}(\beta)$, is formally defined as:

$$\mathcal{C}(\beta) = \{c \in V^* \ : \ s(c, p) = start(\beta), \ (c, p, s(c, p)) \in S\}$$

6.4 The *backwards* phase: the new allocation

At this stage, one of the main objectives is to reallocate the supertasks to m processors. We wish to implement this reallocation process in such a way that the degree of parallelism is maximised. In order to do so, Brent's Principle is applied and therefore, there is the need to determine potential parallel tasks.

Due to the architectural characteristics of the target machine, the superdag was constructed during the first stage on the attempt to bundle messages. The supertasks were scheduled in well-defined intervals of time and the tasks must not be unclustered. At the *backwards* phase, parallel supertasks are identified in order to apply Brent's principle. This fact shows the importance of the regularity on the definition of the bands. Without this regularity, it would be difficult to identify parallel supertasks with the consequence that Brent's Principle could not be properly applied considering our target model.

Each *iteration* of the *backwards* phase corresponds to the visits to the parallel supertasks listed in the set $\mathcal{C}(\beta)$. Brent's Principle is applied and the number of supertasks to be reallocated to each one of the m physical processors during an iteration or band β is defined. This number is given by:

$$\eta(\beta) = \left\lceil \frac{|\ \mathcal{C}(\beta)\ |}{m} \right\rceil$$

Since each supertask $c \in \mathcal{C}(\beta)$ is executed during the interval of time associated with the band β, the tasks in c are not split.

During each iteration β of the *backwards* phase, up to $\eta(\beta)$ supertasks from $\mathcal{C}(\beta)$ are selected, merged and allocated to a single processor in the set $\mathcal{P}$. At the end of the current iteration, up to m new supertasks are produced. The remapping mechanism provide not only a new allocation for the supertasks in V^*, but also the definition of a new set of supertasks. Each processor $p_i \in \mathcal{P}$ may be allocated more than one supertask, but, during each band, at most one new supertask is allocated to p_i.

6.4.1 The bands containing supertasks with successors

For some $\beta > 0$, let $c \in \mathcal{C}(\beta - 1)$ and let $c', c'' \in \mathcal{C}(\beta)$ be such that $(c', c), (c'', c) \in E^*$. During iteration $\beta - 1$, the supertask c is reallocated to a processor, say, p_i. Therefore, either c' or c'' should be reallocated to p_i in order to avoid an increase on the number of resulting communication events.

During an iteration β, the list of processors which have already participated in the reallocation process during the previous iteration is consulted in order to reallocate a subset of the common immediate predecessors of c to its processor.

Let $\beta > 0$ be a band in which supertasks in $\mathcal{C}(\beta)$ have immediate successors already reallocated to processors in $\mathcal{P}$ during the previous iteration $\beta - 1$. When a processor, say, $p_j \in \mathcal{P}$, participates in the reallocation process during iteration β, at most $\eta(\beta)$ supertasks in $\mathcal{C}(\beta)$ are reallocated to it. The processor to which a supertask $c \in \mathcal{C}(\beta)$ is reallocated is denoted by $proc(c)$ (c is reallocated to a single processor). Also, it is important to know the list of supertasks in $\mathcal{C}(\beta)$ reallocated to processor p_j during each iteration β. This list is denoted by $slist(p_j, \beta)$ and formally defined as $slist(p_j, \beta) = \{c \in \mathcal{C}(\beta)\ :\ proc(c) = p_j\}$. The first processors to participate in the reallocation process during iteration β

are given in the list of processors $lproc(\beta) = \{p_j \in \mathcal{P} \ : \ slist(p_j, \beta-1) \neq \emptyset\}$. which are those that have participated in the remapping process during the previous iteration. The supertasks in $slist(p_i, \beta)$ reallocated to processor p_i are merged and defined as a *new supertask*.

6.4.2 The *common-immediate-successors* mechanism

The supertasks in $\mathcal{C}(\beta)$ first reallocated to a processor $p_j \in lproc(\beta)$, are the ones with immediate successors in V^* already reallocated to p_j during iteration $\beta - 1$. Let this subset of supertasks be formally defined by:

$$\Gamma(p_j, \beta) = \{c' \in \mathcal{C}(\beta) \ : \ \exists c \in \ slist(p_j, \beta-1) \text{ such that } (c', c) \in E^*\}$$

The principle behind the definition of the set $\Gamma(p_j, \beta)$ is an attempt to produce new supertasks with low degree and consequently, a reasonably low number of communication events is likely to be incurred under our target model. Figure 6.2 illustrate the benefit provided by the selection of $\eta(\beta)$ supertasks from $\Gamma(p_j, \beta)$. In such cases, only one superedge towards the new supertask allocated to p_j during iteration $\beta - 1$ is produced.

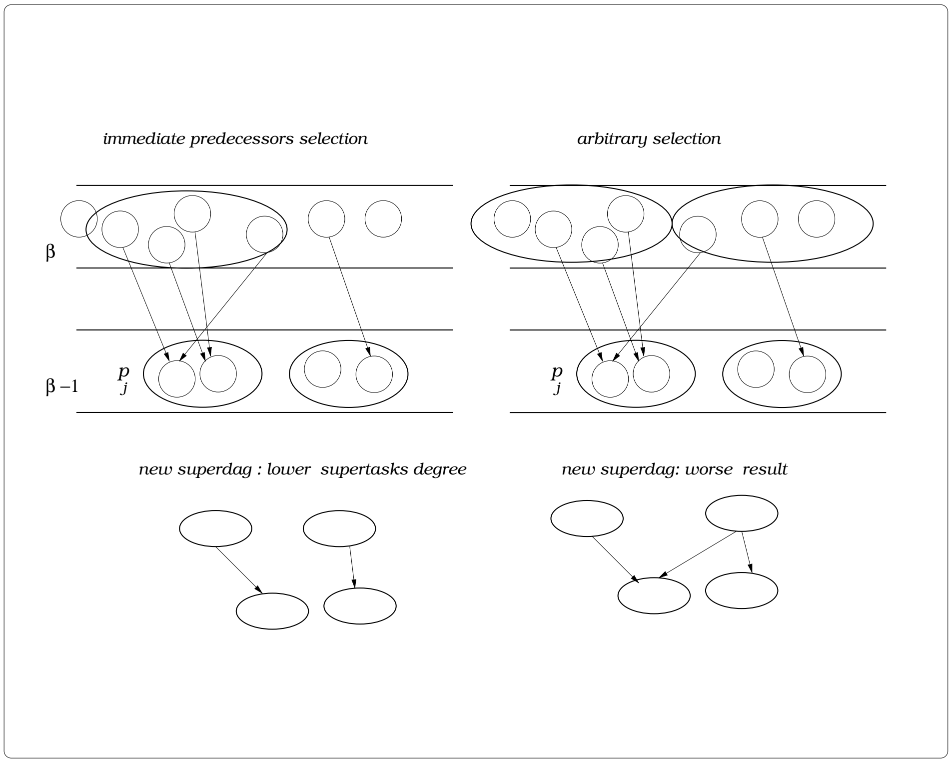


Figure 6.2: The benefit of selecting supertasks with *common-immediate-successors* is shown on the left side. If arbitrary selection is performed, the new supertasks produced may have a higher degree than the one that could be achieved

For each processor $p_j \in lproc(\beta)$, we let $\Gamma(p_j, \beta)$ denote the collection of supertasks first reallocated to p_j during iteration β. The mechanism of selection is called as the *common-immediate-successors* mechanism. Depending on the number of supertasks in the set $\Gamma(p_j, \beta)$, different situations are produced, as discussed below:

$\mid \Gamma(p_j, \beta) \mid = \eta(\beta)$ **:** all immediate predecessors of supertasks in $slist(p_j, \beta - 1)$ are clustered together onto p_j, i.e. all of the supertasks in $\Gamma(p_j, \beta)$ are reallocated to p_j. In this case, the processor p_j is said to be *full*, and no other supertask can be reallocated to it during the iteration β, since the limit dictated by $\eta(\beta)$ must be respected. The processor p_j is then removed from the list $lproc(\beta)$. If there are still processors in $lproc(\beta)$, another processor, say, p_i, such that the list $\Gamma(p_i, \beta)$ is not empty, is selected and the *common-immediate-successor* mechanism is applied again.

$\mid \Gamma(p_j, \beta) \mid < \eta(\beta)$ **:** as in the above case, all the immediate predecessors of supertasks in $slist(p_j, \beta - 1)$ are reallocated to processor p_j, but as a result, p_j is not *full*. Therefore, other supertasks from $\mathcal{C}(\beta)$ may be selected and reallocated to p_j. However, before *filling* the processor p_j, this processor is removed from $lproc(\beta)$, and another processor $p_i \in lproc(\beta)$ such that $\Gamma(p_i, \beta) \neq \emptyset$, is selected, so that the *common-immediate-successor* mechanism can be applied.

$\mid \Gamma(p_j, \beta) \mid > \eta(\beta)$ **:** it is necessary to establish a mechanism to select the group of $\eta(\beta)$ supertasks in $\Gamma(p_j, \beta)$ to be remapped to p_j which leads to the best makespan. This mechanism is described in the next subsection. After selecting this group of supertasks in $\Gamma(p_j, \beta)$ that *fills* processor p_j, the remaining supertasks in $\Gamma(p_j, \beta)$ are ignored for the current moment and another processor $p_i \in lproc(\beta)$ such that $\Gamma(p_i, \beta) \neq \emptyset$ is selected and the *common-immediate-successor* mechanism is applied.

In any case, the supertasks selected in $\Gamma(p_j, \beta)$ are removed from the set $\mathcal{C}(\beta)$, where remain the supertasks yet to be reallocated during the current iteration. Also, in case that p_j is either *full* or partially full, another processor $p_i \in lproc(\beta)$ such that $\Gamma(p_i, \beta) \neq \emptyset$ is selected.

6.4.3 The selection of the immediate predecessors

In order to select the group of $\eta(\beta)$ supertasks in $\Gamma(p, \beta)$, this latter is listed in $\mathcal{L}$ in the decreasing order of the length of messages sent from its supertasks to supertasks in $slist(p, \beta - 1)$ (and thus, adjacent supertasks). Let $k = | \Gamma(p, \beta) |$. Then,

$$\mathcal{L} \;=\; \langle c_1, \ldots, c_k \rangle \text{ where for } i = 1, \ldots, k,\ c_i \in \Gamma(p, \beta) \text{ and where,} \\ c \in slist(p, \beta - 1),\ \text{for } i = 1 \ldots, k - 1, w((c_i, c)) > w((c_{i+1}, c));$$

The reallocation of the first $\eta(\beta)$ supertasks in $\mathcal{L}$ to the same processor p, provides a new superedge produced from the bundling of the longest messages sent by supertasks in $\mathcal{L}$. This new superedge is placed between a new supertask allocated to processor p, during iteration β, and a new supertask allocated to processor p, during iteration $\beta - 1$. The cost of communication associated with the sending of this new message is zero. The reallocation of heaviest-communicating supertasks to the same processor is an effort to produce an efficient schedule, taking into account that the delay parameter τ is also a significant parameter.

However, although supertasks being analysed at the current point are supposed to be reallocated to the processor p, it may be the case that it is necessary to "look" at supertasks already reallocated to processors other than p. Let a supertask, say, c_i, with immediate successors c and c', sends a message x data items long to c and another message x' long to c', such that $x < x'$. Let c and c' be reallocated to different processors p and p', respectively, during previous iterations. Suppose c_i is one of the $\eta(\beta)$ first supertasks listed in $\mathcal{L}$ during iteration β. If c_i is reallocated to processor p, when executing the resulting new superdag G_s in parallel, a delay of $x'\tau$ is incurred due to the sending of the message (c_i, c') (c' is allocated to the other processor p'). On the other hand, if c_i is reallocated to p', a delay of $x\tau$ is incurred due to the sending of the message to processor p, because of the superedge (c_i, c), delay which is a smaller delay than the one in the former case. Consequently, the makespan of the schedule in which c_i is reallocated to p may be greater than the makespan of the schedule in which c_i is reallocated to p' (Figure 6.3).

In order to overcome this unnecessary increase on the delay due to communications a *look-at-others* mechanism is performed. For each supertask $c_i \in \mathcal{L}$, an immediate successor to which c_i sends a maximal-length message is identified; this immediate successor is denoted as c_i^s. Formally,

$$c_i^s : c_i^s \in succ(c_i) \;\wedge\; w((c_i, c_i^s)) = \max_{c' \in succ(c_i)} \{w((c_i, c'))\}$$

If this immediate successor is not in $slist(p, \beta)$ the supertask c_i is removed from the list $\mathcal{L}$. If ties are found, i.e. there is more than one immediate successor of

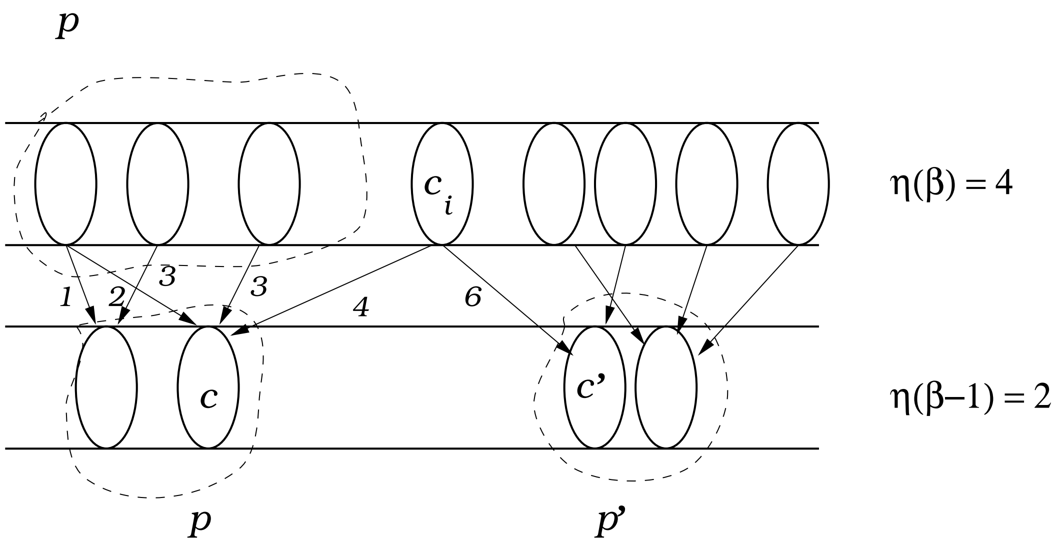


Figure 6.3: Although the supertasks c_i has the higher priority in accordance with the list $\mathcal{L}$ associated with $\Gamma(p,\beta)$, c_i sends a longer message to a supertasks allocated on a processor other than p

$c \in \mathcal{L}$ such as c_i^s, one of which is in $slist(p, \beta - 1)$, then the supertask c_i is not removed from $\mathcal{L}$.

After the evaluation of all of the supertasks in $\mathcal{L}$, the first $\min\{\eta(\beta), \mid \mathcal{L} \mid\}$ supertasks in the resulting ordered list $\mathcal{L}$ are selected and reallocated to p. It may be the case that less than $\eta(\beta)$ supertasks or even no supertask is reallocated to p at this point, and as a consequence, p is not *full*. In the next subsection a mechanism is proposed so that partially full processors may have supertasks reallocated to them.

6.4.3.1 *common-immediate-successors* mechanism applied again?

When initiating the performance of the *common-immediate-successors* mechanism considering a processor $p_j \in lproc(\beta)$, the supertasks in $\mathcal{C}(\beta)$ with common immediate successors, listed in $\Gamma(p_j, \beta)$, are such that $\mid \Gamma(p_j, \beta) \mid > \eta(\beta)$. However, let p_j be such that after applying the *look-at-others* mechanism (which is incorporated in *common-immediate-successors*), we have

$$\mid slist(p_j, \beta) \mid < \ \eta(\beta)$$

This characteristic is due to the fact that the *look-at-others* mechanism applied to the ordered list of supertasks $\mathcal{L}$ associated with $\Gamma(p_j, \beta)$, eliminates so many tasks from $\mathcal{L}$ that there are not enough supertasks left to *fill* the processor p_j.

Let c^* be one of the first $\eta(\beta)$ supertasks in $\mathcal{L}$ which is eliminated from $\mathcal{L}$ (and consequently, from $\Gamma(p_j, \beta)$), due to the *look-at-others* mechanism. Let the immediate successor to which c^* sends the longest message be c', which is reallocated to another processor, say, p_i.

Suppose the situation in which c^* remains in $\mathcal{C}(\beta)$ after the performance of the *common-immediate-successors* mechanism with *look-at-others* incorporated, is applied to **all** processors in $lproc(\beta)$. The supertask c^* is in $\mathcal{C}(\beta)$, because, when *look-at-others* was applied to $\Gamma(p_i, \beta)$, there were $\eta(\beta)$ supertasks in $\mathcal{L}$ which send longer messages to the supertasks in $slist(p_i, \beta-1)$ than the message sent by c^*. Therefore, p_i is *full* without including c^*.

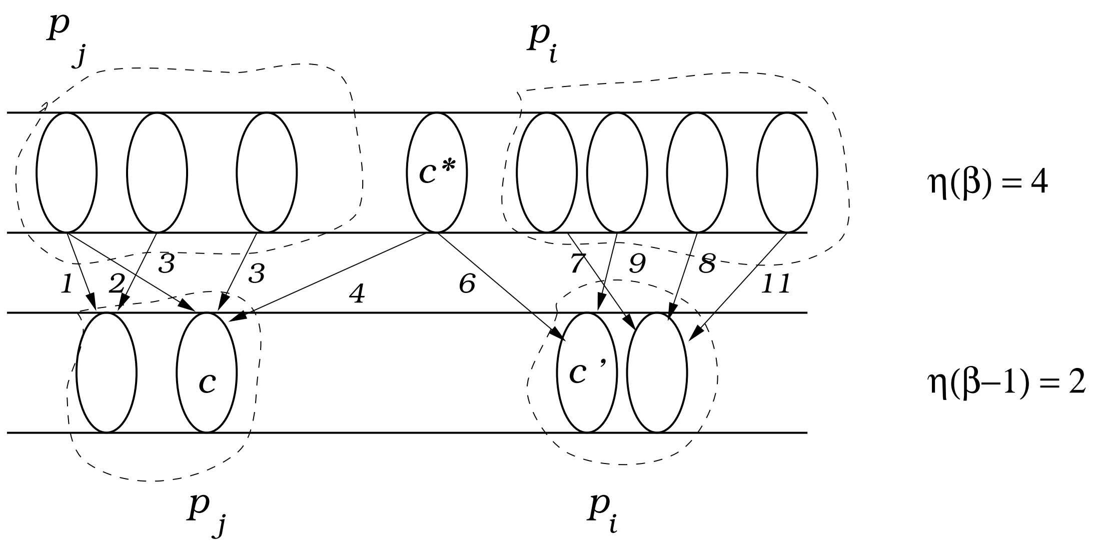


Figure 6.4: A second chance to reallocate supertasks with common successors: the *common-immediate-successors* mechanism is performed again, without incorporating the *look-at-others* mechanism

In order to know where to reallocate supertasks such as c^*, two situations are evaluated:

1. *c^* is reallocated to a processor p_k other than p_i and p_j.* In this case, the delay on the communication of c^* between the two bands β and $\beta - 1$ is proportional to the size of the longest message (c^*, c'). As c^* sends its results to immediate successors allocated to different processors, at least 2 *sending events* are incurred on processor p_k. On each of the processors p_j and p_i, one *receiving event* is incurred due to the sending of messages from the processor executing c^*.

2. *c^* is reallocated to processor p_j* (given that p_i is *full*). In this case, the delay on the communication of c^* between the two bands β and $\beta - 1$ is proportional to the size of the message (c^*, c'). One *sending event* is incurred on processor p_j during iteration β, since c^* sends its results to c'

which is allocated to another processor, and to c. One *receiving event* is incurred on processor p_i, due to the sending of the message (c^*, c').

From the evaluation of the above situations, we conclude that its better to reallocate c^* to processor p_j. In order to do so, it is proposed that, the *common-immediate-successors* mechanism is applied again, but without incorporating the *look-at-others* mechanism.

Therefore, for each processor p_j such that $\mid slist(p_j, \beta) \mid < \eta(\beta)$, we define the list:

$$\Gamma(p_j, \beta) = \{c' \in \mathcal{C}(\beta) \ : \ (c', c) \in E^*, \ \forall c \in \ slist(p_j, \beta - 1)\}$$

Then, the associated ordered list $\mathcal{L}$ is also defined:

$$\mathcal{L} = \langle c_1, \ldots, c_k \rangle : w((c_i, c)) > w((c_{i+1}, c)), \ c_i \in \Gamma(p_j, \beta) \ \wedge \ c \in slist(p_j, \beta - 1)$$

where $k = \mid \Gamma(p_j, \beta) \mid$. Then, the first $\eta(\beta) - \mid slist(p_j, \beta) \mid$ supertasks are selected from $\mathcal{L}$ and reallocated to p_j.

6.4.4 The free allocation

As specified in Subsection 6.4.2, while there exist processors p_j in $lproc(\beta)$ such that the respective set $\Gamma(p_j, \beta)$ is not empty, the *common-immediate-successor* mechanism is applied. However, it is not always the case that there exists the possibility to define a set $\Gamma(p_j, \beta)$, although there are still supertasks in $\mathcal{C}(\beta)$ to be reallocated. This situation arises when:

$\mathcal{C}(\beta) \neq \emptyset \ \wedge \ lproc(\beta) = \emptyset$ **:** all of the processors that had participated during iteration $\beta - 1$, have already participated during iteration β and they are *full*.

$lproc(\beta) \neq \emptyset \wedge \ \nexists p_i \in lproc(\beta) : \Gamma(p_j, \beta) \neq \emptyset$ **:** there still exist processors which are not *full*, but the *common-immediate-successors* mechanism cannot be applied any more. Therefore, any task in $\mathcal{C}(\beta)$ can be reallocated to any processor in $\mathcal{P}$ which is not full.

Thus, there are supertasks in $\mathcal{C}(\beta)$ yet to be reallocated and consequently, there exist processors that are not *full* (due to the definition of $\eta(\beta)$), but no set of tasks with common immediate successors yet to be reallocated can be defined anymore (the set Γ). Each processor $p_i \in \mathcal{P}$ such that $\mid slist(p_i, \beta) \mid < \ \eta(\beta)$ can be selected and assigned *freely* to supertasks in $\mathcal{C}(\beta)$. In order to identify such processors, we formally define

$$free(\beta) = \{p_i \in \mathcal{P} \ : \mid slist(p_i, \beta) \mid < \ \eta(\beta) \ \wedge \ \Gamma(p_i, \beta) = \emptyset\}$$

as the set of partially full or empty processors.

Intuitively, the selection of supertasks to be reallocated on any $p_i \in free(\beta)$ can be done arbitrarily. However, a priority is proposed in order to prevent the specification of new supertasks with relatively high degree (in degree or out degree). The proposed priority establishes that the supertasks to be reallocated to the same processor, say, p_i, are those with a maximal number of *common immediate predecessors*. The application of such a priority is an attempt to avoid a large number of communication events that are incurred when the new superdag is executed in parallel.

The $free\_allocation$ procedure implements the necessary mechanisms to reallocate the remaining supertasks in $\mathcal{C}(\beta)$. Let p_i be a processor selected from $free(\beta)$.

Case 1: The processor p_i may be empty at this point (this can happen when the processor was not used during the previous iteration). A search is carried out for a supertask with a successor already allocated to the processor p_i during earlier iterations. If there is more than one supertask in $\mathcal{C}(\beta)$ with successors already reallocated to p_i, the one that sends the longest message to one of these successors is selected such that heavily communicating supertasks are reallocated to the same processor (although they are in non adjacent bands). In this way, the number of communication events and the delay due to communication may be decreased when executing in parallel the resulting new superdag. Formally, supposing that p_i is a processor in which $slist(p_i, \beta) = \emptyset$ holds, for all $c \in \mathcal{C}(\beta)$ such that $\exists c' \in succ(c)$ where $proc(c') = p_i$, select a supertask c for which

$$w((c, c')) = \max_{c^* \in \mathcal{C}(\beta)} \{ \max_{c'' \in succ(c^*) \, \wedge \, proc(c'') = p_i} \{w((c^*, c''))\}\}$$

We refer to this mechanism as the *successors-same-processor* mechanism. If there is no such supertask c, i.e. there is no task in $\mathcal{C}(\beta)$ which *successors-same-processor* holds, select any supertask from $\mathcal{C}(\beta)$. This arbitrary selection is carried out for the sink supertasks, as well as, for those supertasks reallocated during bands in which more processors are participating in the reallocation process than the number of processors used during iterations where the respective immediate successors were reallocated (earlier iterations than $\beta - 1$).

Case 2: Processor p_i is partially *full* ($\mid slist(p_i, \beta) \mid < \eta(\beta)$), thus a supertask $c \in \mathcal{C}(\beta)$ can be selected such that c has the maximum number of *common immediate predecessors* with supertasks already in $slist(p_i, \beta)$ (i.e. supertasks already reallocated to p_i). This property establishes the *common-immediate-predecessors* mechanism to be applied in such situations. In order to define the mechanism more formally, let

$$P = \bigcup_{c_j \in slist(p_i, \beta)} pred(c_j)$$

be the set immediate predecessors of all supertasks already in $slist(p_i, \beta)$. We let $\mathcal{N}$ denote the maximum number of common immediate predecessors between supertasks already allocated to p_i and each candidate $c \in \mathcal{C}(\beta)$, more formally:

$$\mathcal{N} = \max_{c \in \mathcal{C}(\beta)} \{ \mid pred(c) \cap P \mid \}$$

Finally, the supertask selected is that for which the *common-immediate-predecessor* property was found to hold, i.e. $c \in \mathcal{C}(\beta) \ : \mid pred(c) \cap P \mid = \ \mathcal{N}$. The selected supertask c has the maximum number of immediate predecessors in common with supertasks already in $slist(p_i, \beta)$.

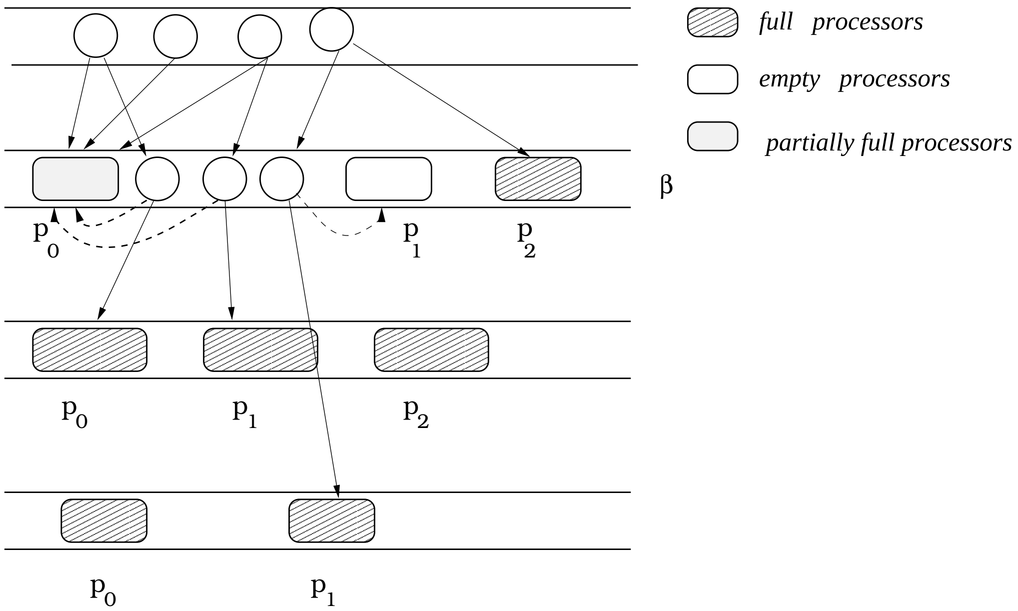


Figure 6.5: During iteration β, after the *common-immediate-successors* mechanism has been applied, the procedure $free\_allocation$ is performed to allocate the remaining supertasks in $\mathcal{C}(\beta)$

However, more than one supertask can have the same number of immediate predecessor supertasks $\mathcal{N}$. In order to break ties, the selected supertask is that which receives the longest message from the common immediate predecessors. Formally specifying, let the list $\mathcal{L} = \{c \in \mathcal{C}(\beta) \ : | \ pred(c) \cap P \ | = \ \mathcal{N}\}$ be the set of all of the supertasks in $\mathcal{C}(\beta)$ with $\mathcal{N}$ *common-immediate-predecessors* with supertasks already reallocated to p_i. Then, the selected supertask c is that in which:

$$\max_{c \in \mathcal{L}} \{\max_{c' \in P} w((c', c))\}$$

If there is no other supertask c such that the *common-immediate-predecessor* property holds, but processor p_i is such that $\mid slist(p_i, \beta) \mid < \eta(\beta)$, then the *successors-same-processor* mechanism is applied on $\mathcal{C}(\beta)$. Again, if this latter property does not hold, select any supertask from $\mathcal{C}(\beta)$.

The selected supertask c is reallocated to p_i, removed from $\mathcal{C}(\beta)$ and finally, included in $slist(p_i, \beta)$. Whenever $\mid slist(p_i, \beta) \mid \geq \eta(\beta)$, the processor p_i is *full* and therefore, it must be removed from $free(\beta)$.

6.4.5 Allocation and the new superdag

During each iteration β, after each group of $\eta(\beta)$ supertasks is selected to be reallocated to processor p_j, a procedure denominated $allocation$ is executed in order to establish the reallocation by constructing the list $slist(p_j, \beta)$. The processor to which each supertask $c \in slist(p_j, \beta)$ is reallocated can be identified by $proc(c)$. The band during which the reallocation of the supertask c is taken place is denoted by $band(c)$. More formally, $proc(c) = p_j, \ \forall c \in slist(p_j, \beta)$ defines the new processor of cluster $c \in V^*$ and $band(c) = \beta, \ \forall c \in slist(p_j, \beta)$ is the band during which the reallocation was taken place. This information is necessary and very useful for the specification of the new superedges, as seen below.

The new supertask is also a cluster of tasks $v_i \in V$. Let $c_{p_j,\beta}$ denote a new supertask allocated to processor p_j during the band β. The new supertask is:

$$c_{p_j,\beta} = \bigcup_{c \in slist(p_j,\beta)} c$$

Note that when merging the supertasks in $slist(p_j, \beta)$, the precedence order among the respective tasks is maintained. The set of tasks in $c_{p_j,\beta}$ corresponds to a new supertask that is included in the set V_s of new supertasks:

$$V_s = V_s \cup \{c_{p_j,\beta}\}$$

The superedges between $c_{p_j,\beta}$ and their immediate successors (also, new supertasks already defined during previous iterations) are specified as follows.

The list of all immediate successors of the supertasks in $slist(p_j, \beta)$ is firstly established:

$$\mathcal{L} = \bigcup_{c \in slist(p_j,\beta)} succ(c)$$

With the identification of a supertask $c \in V^*$, its straight forward to know in which processor it was reallocated ($proc(c)$)and during which band the reallocation was taken place ($band(c)$). With this information, the list of new immediate successor supertasks is formed:

$$\mathcal{L}_{new} = \bigcup_{c \in \mathcal{L}} \{c_{proc(c),band(c)}\}$$

Finally, the superedges are specified, considering the immediate successors of the new supertask $c_{p_j,\beta}$ listed in $\mathcal{L}_{new}$:

$$w((c_{p_j,\beta}, c_{p',\beta'})) = \sum_{\substack{v_i \in c_{p_j,\beta} \\ v_j \in c_{p',\beta'}}} w((v_i, v_j)), \ \forall c_{p',\beta'} \in \mathcal{L}_{new}$$

6.4.6 The end of the *backwards* phase

As the reallocation process initiates from the sink supertasks in G^*, no definitive schedule is produced after the end of the *backwards* phase, but it is provided the new allocation and new *order* of the supertasks in V^* onto m processors, which is represented by the new superdag $G_s = (V_s, E_s)$. At the end of the *backwards* phase, $\mathcal{B}$ bands of parallel new supertasks are identified. However, not necessarily the bands are $\gamma + 1$ wide, since a new supertask is the result of the merge of supertasks reallocated to the same processor. The lower and upper bounds for the execution cost of the new supertasks in V_s can be derived, considering that the supertasks in V^* have execution cost equal to $\gamma + 1$.

Lemma 6.4.1 *Given the schedule S and the superdag G^* produced by the first stage of MSA, considering the clustering factor γ, the execution cost of each new supertask, produced by the backwards phase in the second stage of MSA, is no less than $\gamma + \eta(\beta)$ and no more than $(\gamma + 1) * \eta(\beta)$, for each computational interval β.*

Proof:

***Lower Bound*:** $\eta(\beta)$ supertasks are reallocated to the same processor, say, p_j. All of the supertasks $c \in slist(p_j, \beta)$ were formed by clustering their respective owners together with γ predecessors (specified in the *replication* algorithm). The lower bound on the execution cost of the new supertasks is derived by assuming that the owners of the clusters in $slist(p_j, \beta)$ are tasks with the same set of γ predecessors. Therefore, when merging the clusters in $slist(p_j, \beta)$, as a unique copy of each task in V is maintained, the execution cost of the new supertask is equal to $\gamma + \eta(\beta)$.

Upper Bound: the upper bound on the execution cost of the new supertasks is derived by evaluating the case in which the owners of the clusters in $slist(p_j, \beta)$ are tasks such that their respective set of γ predecessors are disjoint. Therefore, when merging the clusters in $slist(p_j, \beta)$, the execution cost of the resulting supertask is $(\gamma + 1) * \eta(\beta)$.

$\square$

After the iteration β is completed, up to m new supertasks are defined and also, the width of a band β, denoted by $width(\beta)$, is the longest execution cost amongst the new supertasks produced during iteration β:

$$width(\beta) = \max_{p \in \mathcal{P}} \{| \ c_{p,\beta} \ |\}$$

As the precedence relationship among the new supertasks in V_s has already been established, the number of communication events can be calculated. Let $send(c_{p_j,\beta})$ be the number of *sending event* tasks which are incurred on the processor p_j, during the band β, after the supertask $c_{p_j,\beta}$ is executed and the processor sends the respective messages to the immediate successors of $c_{p_j,\beta}$, successors which are not allocated to the processor p_j. Formally,

$$send(c_{p_j,\beta}) \ = | \ \{c_{p,\beta'} \in V_s \ : \ w((c_{p_j,\beta}, c_{p,\beta'})) \neq 0 \ \wedge \ p \neq p_j\} \ |$$

The number of *receiving event* tasks, denoted by $recv(p_j, \beta)$ are incurred on the processor p_j during the band β before the supertask $c_{p_j,\beta}$ is executed. The processor p_j receives messages from all of the immediate predecessors of $c_{p_j,\beta}$ which are not allocated to it. The list of immediate predecessors of $c_{p_j,\beta}$ executed on distinct processors is:

$$recv(c_{p_j,\beta}) \ = | \ \{c_{p,\beta'} \in V_s \ : \ w((c_{p,\beta'}, c_{p_j,\beta})) \neq 0 \ \wedge \ p \neq p_j\} \ |$$

Having set all the above structures and information, the final schedule S_s can be defined. At first, the *backwards phase* procedure is defined, where the bands are visited from the sinks band towards the sources band. An analysis of the supertasks to be reallocated is carried out from lines 7 to 11, within which the *common-immediate-successor* procedure is performed, also performing the *look-at-others* mechanism (the second parameter is set to $TRUE$ so that this mechanism is performed). After all the processors in $lproc(\beta)$ have been analysed, the *common-immediate-successor* procedure is performed again without *look-at-others* mechanism, as specified from lines 12 to 14.

Phase: 9 $Backwards()$ {

1 define $\mathcal{T}(v_i), \ \forall v_i \in V$;
2 $\mathcal{B} := \left\lceil \frac{\mathcal{M}(S)-(2\gamma-1)}{2\gamma+1} \right\rceil$;
3 for $\beta = 0, \ldots, \mathcal{B}-1\{$
4     $start(\beta) := 2((\mathcal{B}-1-\beta) * (\gamma+1) + \gamma) - \gamma$;
5     $\mathcal{C}(\beta) := \{c \in V^* : s(c,p) = start(\beta), \ (c, p, s(c,p), \in S)\}$;
6     $\eta(\beta) := \left\lceil \frac{|\mathcal{C}(\beta)|}{m} \right\rceil$;
7   while $(\mathcal{C}(\beta) \neq \emptyset)$ {
8         $lproc(\beta) = \{p_j \in \mathcal{P} : slist(p_j, \beta-1) \neq \emptyset\}$;
9   while $(lproc(\beta) \neq \emptyset)$ {
10             $p := first(lproc(\beta))$;
11             $common\_immediate\_successor(p, TRUE)$;
        }
12   while $(lproc(\beta) \neq \emptyset)$ {
13             $p := first(lproc(\beta))$;
14             $common\_immediate\_successor(p, FALSE)$;
        }
15         $free\_allocation()$;
    }
}

}

The details of the *common-immediate-successor* mechanism are given in Procedure 10 below. The search for the common immediate successors which receive the longest message is implemented from lines 6 to 10: this is the implementation of the *look-at-others* mechanism. This mechanism is not executed when $LOOK$ is $FALSE$. If the processor is not *full* the necessary supertasks in the list $\mathcal{L}$ associated with the current $\Gamma(p_j, \beta)$ are selected, as seen in lines 11 to 12. The $allocate$ procedure forms the new supertasks and superedges.

Procedure: 10 $common\_immediate\_successor(p_j, LOOK)$ {

$\Gamma(p_j, \beta) \ := \ \{c' \in \mathcal{C}(\beta) \mid \ \exists c \in \ slist(p_j, \beta - 1)$such that $(c', c) \in E^*\}$;
if $(\Gamma(p_j, \beta) \ \neq \ \emptyset)$ {
  if $(\mid \Gamma(p_j, \beta) \mid \leq \ \eta(\beta))$
    $C \ := \ \Gamma(p_j, \beta)$;
  else {
    $\mathcal{L} \ := \ < c_1, \ldots, c_k >$ where for $i = 1, \ldots, k, \ c_i \in \Gamma(p, \beta)$ and where, $c \in slist(p, \beta - 1)$, for $i = 1 \ldots, k - 1$, $w((c_i, c)) > w((c_{i+1}, c))$;
    if $(LOOK)$ {
      $\forall c \in \mathcal{L}$ {   /* *look-at-others* */
        $c_i^s \ : \ c_i^s \ \in \ succ(c_i) \ \wedge \ w((c_i, c_i^s)) \ = \ \max_{c' \in succ(c_i)}\{w((c_i, c'))\}$;
        if $(c_i^s \ \notin \ slist(p_j, \beta))$
          $\mathcal{L} \ := \ \mathcal{L} - \{c_i^s\}$;
      }
    }
    $n \ := \ \min\{\eta(\beta), \mid \mathcal{L} \mid\}$;
    $C \ := $ first n elements of $\mathcal{L}$;
  }
  $allocate(C, p_j)$;
}
}

If there remains supertask in $\mathcal{C}(\beta)$, the $free\_allocation$ procedure is performed. The details of $free\_allocation$ are described in Procedure 11, where the *common-immediate-predecessor* mechanism is implemented from lines 9 to 15. If there are no tasks allocated to the candidate processor during the previous stage, only the *successor-same-processor* mechanism is applied, as specified from lines 5 to 7. In any case, if none of the mechanisms can be applied, any cluster is selected from $\mathcal{C}(\beta)$ (lines 7 or 14).

Procedure: 11 $free\_allocation(C)$ {

1 $free(\beta) \ = \ \{p_i \in \mathcal{P} \ : \ \mid slist(p_i, \beta) \mid < \ \eta(\beta) \ \wedge \ \Gamma(p_i, \beta) = \emptyset\}$;
2 while $((\mathcal{C}(\beta) \ \neq \ \emptyset) \ \wedge \ (free(\beta); \neq \ \emptyset))$ {

p_i is a processor from $free(\beta)$;
while $((\mathcal{C}(\beta) \neq \emptyset) \wedge (|\ slist(p_i, \beta)\ | < \eta(\beta)))\{$
if $(slist(p_i, \beta) = \emptyset)$ $\{$ */* successor-same-processor */*
Let $c' \in \mathcal{C}(\beta)$ be a supertask such that: $w((c, c')) := \max_{c^* \in \mathcal{C}(\beta)}\{\max_{c'' \in succ(c^*) \wedge proc(c'')=p_i}\{w((c^*, c''))\}\}$;
if $\nexists$ such c', select any c' from $\mathcal{C}(\beta)$;
$C := c'$;
$\}$ else $\{$ */* common-immediate-predecessor */*
$P := \bigcup_{c_i \in slist(p_i,\beta)} pred(c_i)$;
$\mathcal{N} := \max_{c \in \mathcal{C}(\beta)}\{\ |\ pred(c) \cap P\ |\ \}$;
$\mathcal{L} = \{c \in \mathcal{C}(\beta)\ :|\ pred(c) \cap P\ | = \mathcal{N}\}$;
Let c' be such that $\max_{c \in \mathcal{L}}\{\max_{c' \in P} w((c', c))\}$;
if $\nexists$ such c', let $c' \in \mathcal{C}(\beta)$ be a supertask such that: $w((c, c')) := \max_{c^* \in \mathcal{C}(\beta)}\{\max_{c'' \in succ(c^*) \wedge proc(c'')=p_i}\{w((c^*, c''))\}\}$;
if $\nexists$ such c', select any c' from $\mathcal{C}(\beta)$;
$C := c'$;
$\}$
$allocate(C, p_i)$;
$free(\beta) := free(\beta) - \{p_i\}$;
$\}$
$\}$

The $allocate$ procedure, described in details below, defines the new supertask and its allocation after the necessary evaluations have been carried out over the parallel supertasks during band β.

Procedure: 12 $allocate(C, p_j)$ $\{$

/ C is the set of clusters to be reallocated to processor p_j */*
$slist(p_j, \beta) := slist(p_j, \beta) \cup C$;
$proc(c) := p_j, \forall c \in slist(p_j, \beta)$;
$band(c) := \beta, \forall c \in slist(p_j, \beta)$;
$c_{p_j,\beta} := \bigcup_{c \in slist(p_j,\beta)} c$;
$V_s := V_s \cup \{c_{p_j,\beta}\}$;
$\mathcal{L} := \bigcup_{c \in slist(p_j,\beta)} succ(c)$;
$\mathcal{L}_{new} := \bigcup_{c \in \mathcal{L}} \{c_{proc(c),band(c)}\}$;

$w((c_{p_j,\beta}, c_{p',\beta'})) := \sum_{v_i \in c_{p_j,\beta}, v_j \in c_{p',\beta'}} w((v_i, v_j)),\ \forall c_{p',\beta'} \in \mathcal{L}_{new};$

$E_s := E_s \cup (c_{p_j,\beta}, c_{p',\beta'}) \forall c_{p',\beta'} \in \mathcal{L}_{new};$

$\mathcal{C}(\beta) := \mathcal{C}(\beta) - slist(p_j, \beta);$

}

6.5 The *updating* phase: final schedule times

Finally, the *updating* phase is executed to define the final schedule S_s for the new superdag G_s, considering the parameters λ_s, λ_r and τ.

The characteristics of the new superdag $G_s = (V_s, E_s)$ that are necessary for the definition of S_s are: the execution time of the new supertasks, the weight associated with the new superedges, the out degree and in degree of the new supertasks. These characteristics, added to the knowledge of the allocation of the new supertasks the fully connected network with m processors, provide enough information for the definition of S_s.

The schedule S_s is characterised by the presence of *bands* of parallel new supertasks and the respective communication event tasks associated with them. During each band β, a processor is busy with the execution of tasks and supertasks in the order given below:

1) *receiving event* tasks associated with the set $c_{p,\beta},\ \forall p \in \mathcal{P}$;
2) the set of parallel new supertasks $c_{p,\beta},\ \forall p \in \mathcal{P}$;
3) the *sending event* tasks associated with the set $c_{p,\beta},\ \forall p \in \mathcal{P}$;

The synchronous patterns found in the resulting schedule S_s is due to the specification of $\mathcal{B}$ bands of parallel new supertasks. During the *updating* phase, in order to calculate the scheduling time of the new supertasks in V_s, the bands are visited from the band $\mathcal{B} - 1$ towards band 0.

An interval of time is associated with the band β, comprising not only the execution time of the supertasks $c_{p_i,\beta}$, for each $p_i \in \mathcal{P}$, but also, their respective communication event tasks, since the overheads associated with the communication events are also included in the final schedule S_s. Therefore, these overheads are also incorporated to $width(\beta)$. For $\beta = \mathcal{B} - 1, \ldots, 0$, the $width(\beta)$ is redefined as:

$$width(\beta) = \max_{p \in \mathcal{P}} \{ \mid (c_{p,\beta}) \mid \ + send(c_{p,\beta}) * \lambda_s + recv(c_{p,\beta}) * \lambda_r \}$$

Note that the width of the band defined above in an *Upper Bound* of the interval of time associated with the respective band.

An interval of time may exist between each pair of bands due to the delay on communications among new adjacent supertasks allocated to distinct processors. The interval of time between each pair of bands β and $\beta - 1$ associated with the delay on the communication is denoted by $delay(\beta, \beta - 1)$. Note that, there may exist an immediate predecessor of a supertask $c_{p,\beta}$ which not necessarily is executed during the band $\beta + 1$. Suppose that this immediate predecessor new supertasks is formed during band β'. In such case, an evaluation on the communication cost associated with the send of the respective message is carried out. The delay on sending the respective message is calculated and compared with the time elapsed since the transmission is initiated, after execution of β', until the time when processors of the previous band $\beta + 1$ complete their execution. Let $c_{p,\beta}$ be a new supertask, as seen in Figure 6.6. One of its immediate predecessors is executed during band $\beta' = \beta + 2$. In the example, the delay on the communication is greater than the finish time of the band $\beta + 1$, therefore, this delay may affect the interval $delay(\beta + 1, \beta)$. On the other hand,

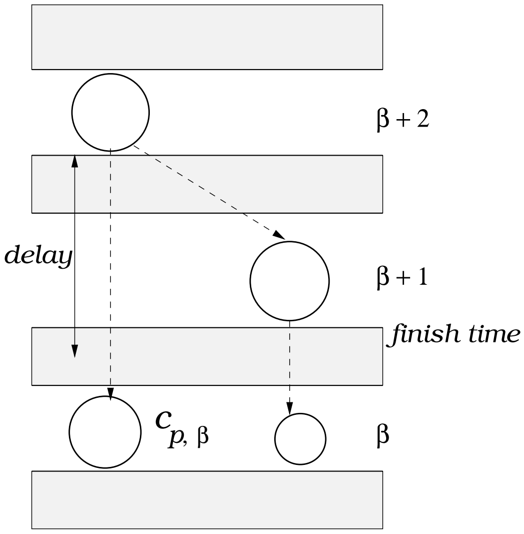


Figure 6.6: The influence of the delay on the communication when communicating supertasks are not executed during adjacent bands. The dashed arrow represents the precedence relationship and the double arrow, the delay time when communicating.

if the finish time of the previous band $\beta + 1$ is greater than the delay due to the communication, this delay does not affect the execution of the band β (and also, the supertask $c_{p,\beta}$).

Let $c_{p',\beta'} \in pred(c_{p,\beta})$. The total elapsed time since the new supertasks and the *communication event* tasks during band β' have their execution completed until the new supertasks and *communication event* tasks band $\beta+1$ are completed is:

$$\mathcal{E}(\beta',\beta+1)=\sum_{i=\beta'}^{\beta+2} delay(i,i-1)+width(i-1)$$

where $\beta' > \beta$. The communication delay due to the sending of the message $(c_{p',\beta'},c_{p,\beta})$ is:

$$\mathcal{D}((c_{p',\beta'},c_{p,\beta}))=w((c_{p',\beta'},c_{p,\beta}))*\tau$$

The message starts to be sent after the execution of the new supertasks and respective communication event tasks. Let $D(c_{p',\beta'},c_{p,\beta})$ represent the delay and the elapsed time that affects the start time of $c_{p,\beta}$. Then,

$$D(c_{p',\beta'},c_{p,\beta})=\begin{cases}\mathcal{D}((c_{p',\beta'},c_{p,\beta}))-\mathcal{E}(\beta',\beta+1) & \text{if } \mathcal{D}((c_{p',\beta'},c_{p,\beta}))>\mathcal{E}(\beta',\beta+1)\\ 0 & \text{otherwise}\end{cases}$$

Finally, for every pair of bands $(\beta+1,\beta),\ \beta=\mathcal{B}-2,\ldots,0$, the delay is defined as:

$$delay(\beta+1,\beta)=\max_{p\in\mathcal{P}}\{\max_{c_{p',\beta'}\in pred(c_{p,\beta})} D(c_{p',\beta'},c_{p,\beta})\}\}$$

The details of the *updating phase* are given in Procedure 13 below.

Phase: 13 $updating()$ {

$\quad\mathcal{B} := \left\lceil\frac{\mathcal{M}(S)-(2\gamma+1)}{2\gamma+1}\right\rceil;$

$\quad\forall\beta=\mathcal{B}-1,\ldots,0$

$\qquad width(\beta) := \max_{p\in\mathcal{P}}\{\mid(c_{p,\beta})\mid +send(c_{p,\beta})*\lambda_s+recv(c_{p,\beta})*\lambda_r\};$

$\quad\forall\beta=\mathcal{B}-1,\ldots,0$ {

$\qquad delay(\beta+1,\beta) := 0;$

$\qquad\forall p\in\mathcal{P}$ {

$\qquad\quad$ if $(slist(p,\beta)\neq\emptyset)$

$\qquad\qquad\forall c_{p',\beta'}\in pred(c_{p,\beta})$

$\qquad\qquad\quad$ if $(p'\neq p)$ $calculate\_delay();$

$\qquad\qquad\quad$ else $D(c_{p',\beta'},c_{p,\beta}) = 0;$

$\qquad$ }

$\quad delay(\beta,\beta-1) := \max_{p\in\mathcal{P}}\{\max_{c_{p',\beta'}\in pred(c_{p,\beta})} D(c_{p',\beta'},c_{p,\beta})\}\};$

$\quad$ }

}

Procedure: 14 $calculate\_delay()$ {

$$\mathcal{E}(\beta', \beta - 1) := \sum_{i=\beta'}^{\beta+2} delay(i, i-1) + width(i-1);$$

$$\mathcal{D}((c_{p',\beta'}, c_{p,\beta})) := w((c_{p',\beta'}, c_{p,\beta})) * \tau;$$

$$D(c_{p',\beta'}, c_{p,\beta}) := \begin{cases} \mathcal{D}((c_{p',\beta'}, c_{p,\beta})) - \mathcal{E}(\beta', \beta - 1) & \text{if } \mathcal{D}((c_{p',\beta'}, c_{p,\beta})) > \mathcal{E}(\beta', \beta - 1) \\ 0 & \text{otherwise} \end{cases}$$

}

6.5.1 Complexity analysis of the *remapping* algorithm

The *remapping* algorithm is analysed by considering each of its phases in turn. The *backwards phase* is characterised by the analysis of each band β defined in line 2. Initially, this first loop visits the parallel supertasks scheduled during each band β. After calculating the number of supertasks to be remapped per physical processor, the loop in line 7 visits each of these supertasks. Therefore, both loops are performed $\mathcal{O}(n)$ times during which all the supertasks in G' are visited. Recall that G' is the superdag produced by the first stage of MSA and, in the worst case, it may have as many supertasks as there are tasks in the input *DAG* G. For each band β, $\mathcal{C}(\beta)$ is defined, which has a maximum that is related to the width $\mathcal{W}(G)$ of the input *DAG* G. The loop in line 9 has $\mathcal{O}(m)$ iterations, one for each processor in $lproc(\beta)$. During each iteration, the $common\_immediate\_successor$ mechanism with *look-at-others* is performed, the analysis for which is as follows.

As a simplification, let us assume $\omega = \mathcal{W}(G)$. During the execution of the $common\_immediate\_successor$ mechanism with *look-at-others*, the list $\Gamma(p_j, \beta)$ is constructed with no more than ω elements. In line 5 of Procedure 10, the list $\mathcal{L}$ is ordered in $\mathcal{O}(\frac{\omega^2}{m} \log \frac{\omega^2}{m})$ steps. This is due to the fact that $slist(p, \beta - 1)$ has $\mathcal{O}(\frac{\omega}{m})$ supertasks and thus, $\mathcal{O}(\frac{\omega^2}{m})$ edges are inspected. The *look-at-others* mechanism is implemented in $\mathcal{O}(\frac{\omega}{m})$ steps, where the search for c_i^s in line 7 takes $\mathcal{O}(n)$ to be executed, given that a supertask can have at most $n - 1$ immediate successors. As a result, the $common\_immediate\_successor$ mechanism is executed in $\max\{\mathcal{O}(\omega n), \mathcal{O}(\frac{\omega^2}{m} \log \frac{\omega^2}{m})\}$ steps. It is easy to see that this mechanism without *look-at-others* is implemented in $\mathcal{O}(\frac{\omega^2}{m} \log \frac{\omega^2}{m})$ steps. As a remark, the width of a *DAG* can be no more than n, i.e. $\omega \leq n$, thus reducing the complexity result to $\max\{\mathcal{O}(n^2), \mathcal{O}(\frac{n^2}{m} \log \frac{n^2}{m})\}$.

On analysing the $free\_allocation$ procedure, line 2 of $free\_allocation$ is executed in $\min\{\mathcal{O}(\omega), \mathcal{O}(m)\}$ steps. At the same time, the loop implemented in line 4 of this procedure can be implemented in $\min\{\mathcal{O}(\omega), \mathcal{O}(\frac{\omega}{m})\} = \mathcal{O}(\frac{\omega}{m})$ steps. Note that, if $\omega \geq m$ then each processor is used and therefore, the first loop (line 2) has $\mathcal{O}(m)$ iterations. On the other hand, if $\omega \leq m$ then there is no need to use all of the processors. Clearly, the worst case occurs when all of the processors are used. The *successor-same-processor* mechanism is implemented in no more than $\mathcal{O}(\omega n)$ steps, given that each of the supertasks in $\mathcal{C}(\beta)$ has at most $n-1$ immediate successors. Due to line 13, the *common-immediate-predecessor* mechanism is also implemented in $\mathcal{O}(\omega n)$ steps. As a result, the complexity of the $free\_allocation$ procedure is $\mathcal{O}(\omega^2 n)$. Taking into account the fact that $\omega \leq n$, then $free\_allocation$ has complexity $\mathcal{O}(n^3)$.

From this, the complexity of the *backwards phase* can be derived to be $\max\{\mathcal{O}(n^3 m), \mathcal{O}(n^3 \log \frac{n^2}{m}), \mathcal{O}(n^4)\}$.

Finally, the *updating phase* is evaluated considering the number of bands $\mathcal{B}$ and the number of processors m. The $calculate\_delay$ procedure is $\mathcal{O}(\mathcal{B})$. A new supertask can have no more than $\mathcal{O}(\mathcal{B}m)$ immediate predecessors. Concluding, the complexity of *updating phase* is $\mathcal{O}(\mathcal{B}^3 m^2)$. Given that the number of bands $\mathcal{B}$ cannot be more than the number of tasks n, the complexity of the *updating phase* is $\mathcal{O}(n^3 m^2)$.

6.6 On the analysis of the number of processors

Typically, *clustering* techniques group tasks without taking into account the number of physical processors. Similarly, the first stage of MSA also does not take into account the actual number of physical processors and therefore, the second stage is required (the *remapping* algorithm). The definition of computational intervals, where sets of parallel supertasks are scheduled, allows the application of Brent's Principle to the superdag G^* producing a schedule on m physical processors. This schedule is provided in an efficient manner, regarding the communication events which are carried out.

When analysing regular *DAG*s (as we will see in Chapter 9), "uniform mapping patterns" can be observed, permitting us to derive the best number of processors to be given to the *remapping* algorithm. This analysis allows us to find the smallest number of processors that can perform the computation with the minimal makespan, given the overall partitioning strategy. Unlike the case of regular *DAG*s where the number of processors may be derived analytically, the number of processors that provides the best schedule for irregular *DAG*s may be difficult to derive automatically.

Many scheduling strategies, such as list scheduling [41, 65], accept the number of physical processors as input. However, a heuristic need not necessarily use all of the given physical processors, since the schedule generated depends on some cost function that guides the scheduling process. Also, the analysis in Section 3.4 shows that there are cases when a smaller number of processors may provide a smaller makespan [54], since such techniques implement locality-exploiting decisions to produce the required schedules.

In the case of the *remapping* algorithm, the m processors might be always used in the reallocation process, since Brent's Principle maximises the exploitation of the degree of parallelism. An iterative execution of the *remapping* algorithm, varying the number of processors, is an opportunity to search for the number of processors that produces the best makespan, given the application and the communication parameters. No more processors than the width of the superdag G^* is necessary, since Brent's Principle is applied. Let $\mathcal{W}(G^*)$ be the width of G^*. Then, the *remapping* algorithm is executed for $m = 1$ to $\mathcal{W}(G^*)$ and the schedule with the smallest makespan is the best one that MSA can provide.

We wish now to give an insight on the characteristics that may affect the specification of the best number of processors. At first, we consider the model in which the $delay$ parameter τ is the only significant parameter. After that, the CLAUD parameters are considered when analysing primitive structures like fork and join.

6.6.1 Delay is the sole parameter

Let $G^* = (V^*, E^*)$ be the superdag produced by the first stage of MSA and let $\mathcal{B}$ be the number of *computational* intervals in S^*. Suppose that there exists at least one band with $\mathcal{W}(G^*)$ parallel supertasks and the other bands have at most $\mathcal{W}(G^*) - 1$ parallel supertasks. After the *remapping* algorithm, the number of communication intervals is decreased when the following situation occurs. Suppose that two bands β and β' are such that the supertasks in β are remapped before the supertasks in band β'. Suppose that m processors are used by the supertasks in $\mathcal{C}(\beta)$. Suppose, w.l.o.g., that c and c' are the new supertasks remapped during β band. If the immediate predecessors of c and c' are in $\mathcal{C}(\beta')$ and the new supertasks produced during band β' are c_i and c'_i such that the new superedges are (c, c_i) and (c', c'_i), then, no inter-processor communication is carried out since at its outset, the *backwards* phase reallocates supertasks to the same processor as their immediate successors.

However, it is not always the case that supertasks are remapped and merged in such a way that inter-processor communication events are not incurred. On the evaluation of the impact of the number of processors given to the *remapping* algorithm, two situations are considered: one in which $\mathcal{B} - 1$ communication intervals are produced by the *remapping* algorithm and the other, in which $\mathcal{B} - 2$ communication intervals are defined. In this section, the makespan of the schedule provided by the *remapping* algorithm for the given number of processors m is denoted by $\mathcal{M}_m$.

Case 1) $\mathcal{B} - 1$ communication intervals :

Suppose that for any $m < \mathcal{W}(G^*)$, the respective result provided by the *remapping* algorithm contains $\mathcal{B} - 1$ communication intervals, that is, supertasks are not merged into a new supertask in such a way that inter-processor communications occur between any pair of adjacent bands.

Let $m = \mathcal{W}(G^*) - 1$. Suppose that β' is the only band with $\mathcal{W}(G^*)$ supertasks and each band $\beta \neq \beta'$ has less than or equal to $\mathcal{W}(G^*) - 1$ parallel supertasks. For all of the bands $\beta \neq \beta'$, the computation time associated with the execution of the new parallel supertasks scheduled to each one of these bands is $\gamma + 1$. This is due to the fact that the number of supertasks reallocated per processor is

$$\left\lceil \frac{\mathcal{C}(\beta)}{m} \right\rceil \leq \left\lceil \frac{\mathcal{W}(G^*) - 1}{\mathcal{W}(G^*) - 1} \right\rceil = 1$$

where $\mathcal{C}(\beta)$ is the number of parallel supertasks yet to be remapped during each band β.

The number of data items sent to the new supertasks scheduled during all of the bands $\beta \neq \beta'$ is no more than $(\gamma + 1)$ data items and thus, communication cost associated with the sending of the respective data items is no more than $(\gamma + 1)\tau$ (the length of the communication interval). When reallocating the parallel supertasks during band β', the number of supertasks reallocated per processor is

$$\left\lceil \frac{\mathcal{W}(G^*)}{\mathcal{W}(G^*) - 1} \right\rceil = 2$$

As shown in Lemma 6.4.1, the execution cost of such new supertask is at least $\gamma + 2$. Consequently, the number of data items sent by the new supertasks in band β' is no less than $\gamma + 2$. Thus, the makespan of the schedule produced by the *remapping* algorithm considering $m = \mathcal{W}(G^*) - 1$, is at least

$$\mathcal{M}_m = (\mathcal{B} - 1)(\gamma + 1) + (\mathcal{B} - 2)(\gamma + 1)\tau + (\gamma + 2) + (\gamma + 2)\tau \qquad (6.1)$$

On the other hand, if $\mathcal{W}(G^*)$ processors are used, the number of supertasks reallocated to each processor during band β' is also equal to 1 and the execution time of these new supertasks is $\gamma+1$. Consequently, the new supertasks (which coincide with the "old: supertasks) send no more than $(\gamma + 1)$ data items. The makespan of the produced schedule is thus

$$\mathcal{M}_{\mathcal{W}(G^*)} = \mathcal{B}(\gamma + 1) + (\mathcal{B} - 1)(\gamma + 1)\tau \qquad (6.2)$$

The difference $\mathcal{M}_m - \mathcal{M}_{\mathcal{W}(G^*)}$ is:

$$\mathcal{M}_m - \mathcal{M}_{\mathcal{W}(G^*)} = 1 + \tau$$

which is non-negative for any value of τ. Therefore, in the case in which adjacent supertasks are reallocated such that interprocessor communication is carried out, the best number of processors to be used is the width of G^*.

Note that if the upper bound (derived in Lemma 6.4.1) on the execution costs of the new supertasks in band β' is considered, the resulting makespan $\mathcal{M}_m$ would be higher.

Case 2: less than $\mathcal{B} - 1$ communication intervals :

Now we observe that there exists $m < \mathcal{W}(G^*)$ given to the *remapping* algorithm in which the number of resulting communication intervals is less than $\mathcal{B} - 1$. Suppose that $m = \mathcal{W}(G^*) - 1$ and that there exists at least one band β' where $\mid \mathcal{C}(\beta') \mid = \mathcal{W}(G^*)$ and for all $\beta \neq \beta'$, $\mid \mathcal{C}(\beta') \mid \leq m$. Thus, the number of supertasks reallocated to each processor during β' is $\eta(\beta') = 2$. There exists at least one processor, say, p, which reallocates 2 supertasks, producing a new supertask, say, c. The upper bound on the execution costs of c (Lemma 6.4.1) is $\eta(\beta')(\gamma+1) = 2(\gamma+1)$. Therefore, for $m = \mathcal{W}(G^*)-1$, the makespan of the schedule where $\mathcal{B}-2$ *communication* intervals are defined is $\mathcal{M}_m = 2(\gamma + 1) + (\mathcal{B} - 1)(\gamma + 1) + (\mathcal{B} - 2)(\gamma + 1)\tau$, or simply

$$\mathcal{M}_m = (\mathcal{B} + 1)(\gamma + 1) + (\mathcal{B} - 2)(\gamma + 1)\tau \qquad (6.3)$$

Note that, since β' is the band in which supertasks are merged, the communication interval between β' and $\beta' - 1$ is the one that "disappears" and no inter-processors communication is carried out. Comparing Equations 6.2 and 6.3, we have that

$$\mathcal{M}_m - \mathcal{M}_{\mathcal{W}(G^*)} = (\gamma + 1)(1 - \tau)$$

Therefore, we conclude that for $\tau \geq 2$, then:

$$\mathcal{M}_m < \mathcal{M}_{\mathcal{W}(G^*)}$$

6.6.1.1 Concluding remarks

The above evaluation aims to illustrate the difficulties and dependencies when deriving analytically the best number of processors to be given to the *remapping* algorithm considering an arbitrary *DAG*. The number of communication intervals that are produced by the second stage depends on the structure of the superdag G^*, which depends on the *clustering factor* γ and G. If there is an automatic way to know that $\mathcal{B} - 1$ *communication* intervals are produced by the *remapping* algorithm, then the best number of processors is the width of the superdag G^*. On the other hand, if less communication intervals are produced, the evaluation above does not necessarily show which number of processors should be given, since the characteristics of the resulting superdag G_s may vary with m.

6.6.2 Considering overheads and delay parameters

The presence of the overhead parameters, increase the complexity of the evaluation mainly in irregular *DAG*s. An irregular *DAG* can be seen as a set of fork and join structures, as considered in [42, 46, 74]. However, we shall see that, in the attempt to derive the number of processors that leads to the best scheduled produced by our strategy, the whole *DAG* must be analysed. Nevertheless, on the evaluation of the impact of the number of processors given to the *remapping* algorithm when considering λ_s, λ_r and τ, the simplest structures are analysed: the join and fork *DAG*s.

6.6.2.1 The join *DAG*

The join *DAG* (Figure 6.7) is affected mostly by the *receiving event* cost λ_r. When the delay is the only significant parameter (the overheads are negligible), the optimal schedule is achieved in the manner described in [47]. Let the $n-1$ source tasks be listed in $V_{order} = \langle v_0, v_1, \ldots, v_{n-2}\rangle$ such that $m(v_0) + w((v_0, v_{n-1})) \geq m(v_1) + w((v_1, v_{n-1})) \geq \ldots \geq m(v_{n-1}) + w((v_{n-1}, v_{n-1}))$, where v_{n-1} is the sink task. The tasks allocated to the same processor are $\langle v_0, \ldots, v_k\rangle \subseteq V_{order}$, for some $k \leq n-1$, such that

$$\sum_{i=0}^{i=k} m(v_i) \leq m(v_k) + w((v_k, v_{n-1}))$$

While all tasks in $\{v_1, v_2, \ldots, v_k\}$ are being executed on the same processor, say, p, communication is being carried out between v_i and v_{n-1}, $i = k+1, \ldots, n-2$, each one of these tasks executed on a distinct processor.

However, when overheads are incurred, the above scheme may not provide the best allocation of the tasks (or supertasks) since, while one processor is executing k tasks, one *sending event* task is scheduled in each one of the other processors, and $n-k$ *receiving events* are incurred on processor p allocated to v_{n-1}, before this task is executed.

In order to decrease this number of communication events, Brent's Principle is applied to "evenly" allocate tasks onto the m processors. Let S_{join} be the scheduled produced. Suppose that each task has execution cost equal to $\gamma + 1$ and that the number of tasks reallocated to each of the m processors is $\lceil\frac{n-1}{m}\rceil$ (as a simplification, let $\lceil\frac{n-1}{m}\rceil = \frac{n-1}{m}$), each processor starts executing its $\frac{n-1}{m}$ tasks at time 0, finishing at time $(\gamma+1) * \frac{n-1}{m}$. After that, a single *sending event* is carried out in each one of the processors during λ_s time steps. Then, the messages are transmitted taking $w\tau$ times steps, where w is the size of the maximal-cardinality message. The message size w also depends on the number of processors m. The weight w results from the bundling of $\frac{n-1}{m}$ edges (or superedges) of the original join *DAG*. Supposing that the edges have the same weight w', then, we have that $w = w'\frac{n-1}{m}$. Finally, before executing the sink task, the processor p receives all of the messages, and therefore, $m-1$ *receiving events* are incurred. Concluding, the makespan of the schedule S_{join} is then,

$$\mathcal{M}(S_{join}) = (\gamma+1)\frac{n-1}{m} + \lambda_s + w'\frac{n-1}{m}\tau + (m-1)\lambda_r + \gamma + 1$$

The number m_{opt} of processors which produces a schedule with minimal makespan is given by calculating $\frac{\delta \mathcal{M}(S_{join})}{\delta m} = 0$. Therefore,

$$m_{opt} = \sqrt{\frac{[(\gamma+1) + w'\tau](n-1)}{\lambda_r}}$$

The allocation of a join *DAG* taking into account the overheads and delay parameters is illustrated in Figure 6.7.

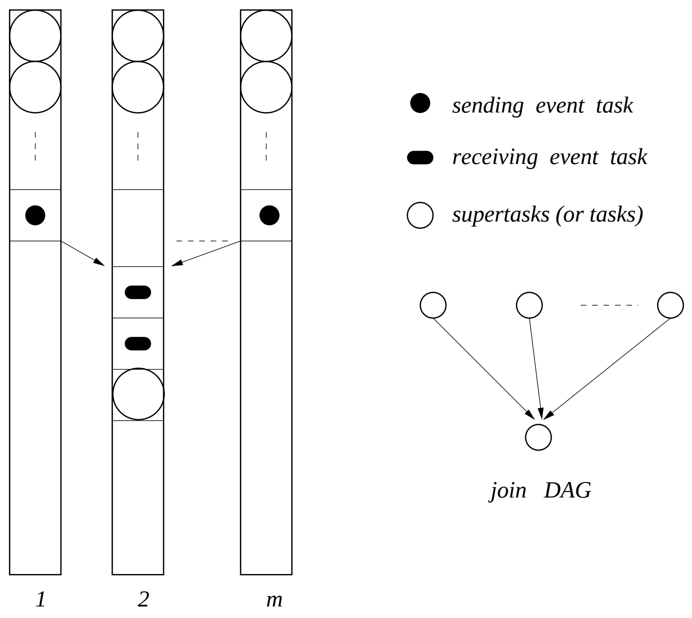


Figure 6.7: The join *DAG* and an allocation considering the overhead parameters and delay: the tasks are evenly distributed between the processors.

The above evaluation is accurate, for the join *DAG* produced by the first stage of MSA, if the upper bound on the execution costs of the *new supertasks* produced by the second stage is assumed (Lemma 6.4.1). Considering the lower bound, the makespan produced is then,

$$\mathcal{M}(S_{join}) = \gamma + \frac{n-1}{m} + \lambda_s + w'\frac{n-1}{m}\tau + (m-1)\lambda_r + \gamma + 1$$

and consequently, the number m_{opt} of processors which produces S_{join} with the minimal makespan is

$$m_{opt} = \sqrt{\frac{(1+w'\tau)(n-1)}{\lambda_r}}$$

On the other hand, if a join *DAG* is given as input to the first stage, no replication is necessary, since each task (the sources) have a single immediate successor. Therefore, the input superdag G^* is isomorphic to the join *DAG* and each supertask has unit execution cost and each superedge is associated with a unit weight. Therefore, the makespan of the schedule S_{join} produced by the *remapping* algorithm is

$$\mathcal{M}(S_{join}) = \frac{n-1}{m} + \lambda_s + \frac{n-1}{m}\tau + (m-1)\lambda_r + 1$$

and therefore,

$$m_{opt} = \sqrt{\frac{(n-1)(1+\tau)}{\lambda_r}}$$

We conclude that given a join structure and considering all of the communication parameters of our target model, the best number of processors depends on one of the communication parameters, the λ_r parameter.

6.6.2.2 The fork *DAG*

The same analysis is done considering the fork *DAG* with n tasks (or supertasks), where v_0 is the source task and $v_1, \ldots, v_{n-1}$ are the sink tasks, as shown in Figure 6.8. If the fork *DAG* is given as input to the first stage, obviously, the minimal makespan is achieved when the source task is replicated together with each one of the sink tasks onto distinct processors (one processor for each sink task). Therefore, the number of processors necessary to achieve the minimal makespan is $n-1$. On the other hand, let us evaluate the case in which the fork superdag is produced by the first stage, where each supertask has the same execution cost $(\gamma+1)$. Let w_i be the length of the message sent from supertask v_0 to the sink v_i and let S_{fork} be the schedule produced by the *remapping* algorithm, considering λ_s, λ_r and τ. The source supertask is scheduled on a processor, say, p, at time 0. After $(\gamma+1)$ times steps, $(m-1)$ *sending events* are issued on p, since the $n-1$ immediate successor supertasks are evenly mapped onto m processors.

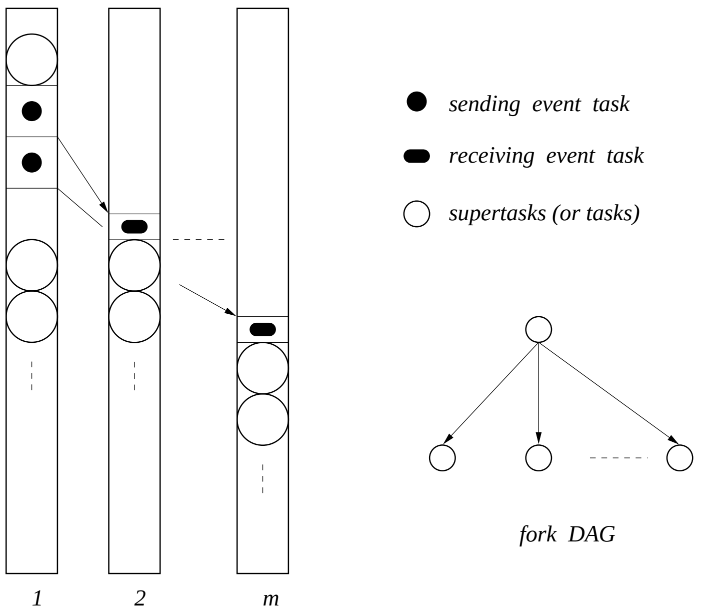


Figure 6.8: The fork *DAG* and an allocation considering the overheads parameters and delay: the tasks are evenly distributed between the processors.

In the case of the fork superdag, the $\frac{n-1}{m}$ messages bundled may not have a weight greater then any one of the original weights of the bundled superedges may have associated common data items. The amount of data sent depends on which superedges are bundled and on m. The $i\underline{\text{th}}$ message is sent after i *sending events* are incurred. As a matter of simplification, suppose that all of the weights are the same and therefore, the last message is transmitted after the $m-1$ *sending events* are incurred. The lower and upper bound derived in Lemma 6.4.1 are considered, and consequently, the respective makespan is given in the two equations below:

Lower Bound: $\mathcal{M}(S_{fork}) = (\gamma+1) + (m-1)\lambda_s + w\tau + \lambda_r + \gamma + \frac{n-1}{m}$

Upper Bound: $\mathcal{M}(S_{fork}) = (\gamma+1) + (m-1)\lambda_s + w\tau + \lambda_r + (\gamma+1)\frac{n-1}{m}$

Consequently, the number m_{opt} of processors that must be given to the *remapping* algorithm in order to produce the minimal makespan is, respectively:

Lower Bound: $m_{opt} = \sqrt{\frac{(n-1)}{\lambda_s}}$

Upper Bound: $m_{opt} = \sqrt{\frac{(n-1)(\gamma+1)}{\lambda_s}}$

As expected, the best makespan for the fork structure depends on λ_s.

6.6.2.3 Some remarks

In [42, 86], Lewis *et al.* propose how to schedule a composite task graph, method which is described and discussed in Chapter 3, Section 3.3.3. In their description, a composite graph is considered to be a collection of regular structure, and the communication parameters defined in the target model are the overhead for sending and the delay. Although there is no discussion about the efficiency of the method, the strategy proposed in [42] initially calculates the lowest number of processors for each regular structure. In order to schedule the whole graph, the total necessary number of processors is the summation of the derived numbers and then, the total number of processors is given to a list scheduling heuristic so that the final schedule is produced.

The possibility to use a similar idea in order to establish the number of processors to be given as input to the *remapping* algorithm becomes difficult when we try to identify regular patterns, like, join and fork structures, on the superdag G^*. Even if it would be possible to identify such structures, not necessarily each fork and join should be treated as an independent structure. Depending on the communication parameter values, it might be better to schedule a group of structures on the same processor. Therefore, the iterative approach used to execute MSA by varying the number of processors when performing the *remapping* algorithm is proposed.

Although the notion of topology has been defined in the CLAUD model, the second stage of MSA does not consider topologies other than the fully connected network. In the rest of this thesis, the evaluation of different approaches on different models take into account a fully connected network. However, as a future work, topology is an issue to be considered and evaluated in the *multi-stage approach*.

6.7 Concluding remarks

The difficulties found to produce a reasonable schedule of general *DAG*s onto a set of m physical processors, considering our target model, are expressed by the number of mechanism specified in the second stage of MSA, the *remapping* algorithm. Difficulties are stressed when parameters like the overheads λ_s and λ_r are considered. Due to their essence, the overhead costs are charged whenever related entities are allocated to distinct locations. Therefore, it is crucial that the relationship between the supertasks (or simply tasks) are evaluated, so that a small number of *communication events* is incurred on the processors allocated to the critical path of the superdag.

In the *remapping* algorithm, a cost function is not explicitly defined in order to guide the allocation of the supertasks, like, for example, the cost function defined in the ETF algorithm. The *remapping* algorithm has as objective to find a reasonable solution to the *max-min* problem. In order to maximise of the degree of parallelism, *parallel supertasks* are identified (produced by the first) and Brent's Principle is applied to *each one of the sets* of parallel supertasks. Therefore, rather than remapping all of the tasks to m processors, the degree of parallelism is maximised. The minimisation of the communication cost is carried out through the mechanisms implemented. The definition of the new supertasks and their new allocation are such that a small number of *communication events* is incurred and messages sent between distinct processors are relatively minimised.

The *common-immediate-successor* mechanism minimises the communication costs by decreasing the number of *communication events*. The *look-at-others* mechanism avoids that the messages sent between distinct processors are the largest one, and therefore, the delay on communication is decreased. In the $free\_allocation$ procedure, the mechanism *successor-same-processor* is an attempt to decrease both the number of *communication events* and to allocate communicating supertasks which exchange maximal-length message to the same processor and therefore, minimising the communication costs. In their essence, all of these mechanisms are evaluating the structure of the input superdag (or *DAG*), that is, the adjacencies among the tasks.

When irregular *DAG*s, the proposed mechanisms are performed with a greater frequency than when regular *DAG*s are used. For example, if the input superdag is a Binary Tree (an in tree in which each supertask, apart from the sink, has out degree equal to one), the only mechanisms that has to be performed is the *common-immediate-successors* mechanism, taken into account that the sink is allocated to an arbitrary processor and that $m = 2^k$, for some integer k. For an irregular *DAG,* this may not be the case and all of the mechanisms may be executed.

Chapter 7

Versatility of the Multi-Stage Approach

7.1 Tuning the strategy for different models

As we have discussed throughout this dissertation, there are communication layers in which the *overheads* due to sending and receiving of messages are much higher than the delay. When devising the first stage of MSA, the main objective was to minimise the number of communication events which are incurred when executing a given application in parallel, due to the characteristics of the target architecture.

Nevertheless, in some parallel architectures, the multicomputer comprises a set of nodes, and inter-node communication is handled by dedicated processors; the other processors execute tasks of the application. When modelling such architecture, the overheads are negligible and the communication cost may be characterised as a delay: task execution can be overlapped with message transmission [9].

Considering these aspects of different architectures, the rules in the $superdag$ algorithm are evaluated and the impact of each rule on different architectural models is analysed. This evaluation examines the influence of λ_s, λ_r and τ on the rules embodied in the $superdag$ algorithm. Therefore, both cases in which $\frac{\lambda_s+\lambda_r}{2} \gg \tau$ and $\tau \gg \frac{\lambda_s+\lambda_r}{2}$ are evaluated. Also, the case where τ is the only significant communication parameter (the overheads are negligible and $\lambda_s = \lambda_r = 0$ is assumed) is analysed and interesting properties are addressed.

7.1.1 The first 3 rules

The *Candidate Clusters Set*, *Restricting the Set* and the *Unique Copy* Rules are necessary for any values of λ_s, λ_r and τ, because these are rules which define the search space of the candidate clusters. The candidate clusters must be defined, since it is important to establish the precedence relationship among the new supertasks. The restriction on the search space aims to make the selection process faster, without specifying a space which is too far from the optimal.

7.1.2 The amount of data sent by the superedges

The importance of the *Maximal Subset Choice* Rule has been already discussed in Section 5.4.4.4, when the underlying model specifies that $\frac{\lambda_s+\lambda_r}{2} \gg \tau$. The implementation of the *Maximal Subset Choice* Rule may lead to a low number of superedges and consequently a low number of communication events may be incurred on the receipt of data by the processors. The total number of superedges is the upper bound for the number of communication events.

On the other hand, when τ is the only predominant parameter, the smaller are the messages (within some limit) transmitted between processors, the smaller is the communication cost. If there is no overhead associated with the sending and receiving of messages, there is no need to be concerned with the total number of superedges defined, since the communication costs are totally dominated by the latency parameter τ.

We conclude that depending on the values of the communication parameters, different mechanisms are needed to produce a good superdag. At next, we discuss how the *Subset Choice* Rule can be parameterised in accordance to each situation.

7.1.2.1 The overheads as critical parameters

The rules in the $superdag$ algorithm specify that a candidate cluster c_i is selected as an immediate predecessor cluster of a visited supertask c, if c_i sends to c the maximal amount of data in $\mathcal{I}'(c)$ (this data has not been sent to c by any other immediate predecessor cluster already selected during $phase_c$).

Suppose that the candidate c_i is selected as immediate predecessor of c during iteration, say, i. Considering the set $\mathcal{I}'(c)$ in the beginning of iteration i, c_i sends to c the data $c_i \cap \mathcal{I}'(c)$ and the set $\mathcal{I}'(c)$ is updated. Let c_j be the candidate selected in the subsequent iteration. This is due to the fact that $c_j \cap \mathcal{I}'(c)$, is the maximal-cardinality subset of $\mathcal{I}'(c)$ in iteration $i+1$. As a consequence, the immediate predecessor clusters are such that they send data items correspondent to disjoint subsets of tasks in $\mathcal{I}(c)$, because at each iteration $\mathcal{I}'(c) \cap c_i$ is removed from $\mathcal{I}'(c)$.

Note that, the superedges incident to each supertask $c \in V^*$ may have different weights. During the transmission of a long message, smaller messages may reach the common destination and the respective receiving overheads are incurred. Therefore, it is possible that the execution of *receiving event* tasks, which are incurred as soon as the smaller messages arrive at the destination processor overlap with the transmission of longer messages.

Lemma 5.4.2 shows that when maximal-cardinality messages are defined, the number of messages received by a supertask is smaller. In a system in which the overheads are such that $\lambda_s > \lambda_r \gg \tau$, the sending of long messages does not necessarily hinder the performance of the resulting schedule, since the overheads are very high. For example, suppose the situation in which l messages are sent by l distinct supertasks, such that $w_1 \geq \ldots \geq w_l$, where w_i is the size of the $i\underline{\text{th}}$ message. Suppose also that these supertasks and the destination supertask are scheduled during two adjacent *computational* intervals β and $\beta + 1$, respectively. Hence, the duration of the communication interval between β and $\beta + 1$ is no less than

$$d \ = \ \lambda_s + w_1\tau + l\lambda_r$$

(each supertask has at least unit out degree and all of them are allocated to distinct processors). Suppose another case: smaller messages are defined so that $l + 1$ immediate predecessor supertasks are necessary. Suppose that the length of the messages is no more than w'. Then, the duration of the communication interval between β and $\beta + 1$ is:

$$d' \ = \ \lambda_s + w'\tau + (l+1)\lambda_r$$

Then, we have that $d' - d \ = \ \lambda_r + (w' - w_1)\tau$. Therefore, situation d' leads to a smaller communication time only if

$$\lambda_r < (w_1 - w')\tau \tag{7.1}$$

Note that $w_1 < \gamma$ and $w' < \gamma$, since a supertask cannot send a message of length greater than the supertask's size. Furthermore, in this case, $w_1 - w' < \frac{w_1}{2}$ and so, $\lambda_r < \frac{w_1}{2}\tau$. However, for the architectures under consideration, $\lambda_r \gg \tau$, i.e. $\lambda_r = \mathcal{X}\tau$ where $\mathcal{X}$ is a huge value. Therefore, this implies that in Equation 7.1, w_1 is a very large value. But all message lengths are smaller that γ. If γ is huge, the application should be effectively executed sequentially.

In summary, when the overheads are predominant and much greater than the delay, the definition of *maximal-cardinality* messages does not hinder the performance of the resulting schedule even if the delay is not totally negligible.

7.1.2.2 The latency is the only predominant parameter

In the case in which the latency τ has a value much greater than the values associated with the overhead parameters, the sending of many packets between two processors may be associated with a large delay when supertasks communicate. If the overheads are negligible, the selection of candidates which send the maximal-cardinality subset of $\mathcal{I}'(c)$ at each iteration may not lead to a good solution. In such a case, a different parameterisation of the *Subset Choice* Rule is proposed.

Given $\Delta'(c)$ and $\mathcal{I}'(c)$, the new mechanism selects the candidate which sends the *minimal-cardinality* message, defining the *Minimal Subset Choice* Rule which is formally defined as follows. As already defined, the number of immediate predecessors in $\mathcal{I}'(c)$ which are in each cluster $c' \in \Delta'(c)$, during $phase_c$ is:

$$ntasks(c', c) \ = \mid c' \cap \mathcal{I}'(c) \mid$$

Let x be the shortest message sent by the candidates in $\Delta'(c)$, i.e.

$$x = \min_{\forall c' \in \Delta'(c)} \{ntasks(c', c)\}$$

The candidates which are maintained in the restricted set are those which contain x tasks from $\mathcal{I}'(c)$. Therefore, given that

$$X = \{c' \in \Delta'(c) \ : \ ntasks(c', c) \neq x\}$$

the restricted set $\Delta'(c)$ is updated to:

$$\Delta'(c) = \Delta'(c) - X$$

As a consequence, the degrees of the supertasks tend to be higher than in the case in which the *Maximal Subset Choice* Rule is executed. However, this is not a problem for the *delay* model and will not affect the effectiveness of the resulting schedule.

Unbundling messages

The superdag concept makes the conservative assumption that all tasks inside each supertask must be executed before any message is sent to its successors and also, these tasks cannot start executing before all of the messages are received from its immediate predecessors. When no overhead is associated with the sending and receiving events, the bundling of messages may lead to a makespan much greater than the one in which messages are not bundled.

Rather than applying another technique in order to find a good schedule for the input application on such kind of systems, MSA is used. After the final schedule on m processors is specified by the *second* stage, a mechanism that *unbundles messages* is implemented. Given the superdag G_s produced by the *remapping* algorithm and its allocation to the m processors, the supertasks are *opened*, allowing the sending of a message (whenever required) after each task is executed. The allocation of the supertasks remains the same, i.e. if a supertask c is allocated to a processor p, then all tasks $v_i \in c$ remains allocated to p after the *unbundled-messages* procedure. It is important to bear in mind that tasks may be replicated and the precedence relation between the supertasks establishes the dependencies between tasks.

When unbundling messages, the schedule time of each task must be recalculated, since tasks can be executed as soon as they receive all the data from immediate predecessors. In order to recalculate the start time of each task, the allocation and the ordering of tasks in each processor are considered. Due to replication, a given task may have an immediate predecessor replicated onto various processors, but the data of only one copy is enough to allow the execution of the task. Let v_j be a task in cluster $c \in V_s$ allocated to processor p_k. Let $v_i \in pred(v_j)$, $c \notin Clusters(v_i, S)$ and $p_k \notin processors(v_i)$. Therefore, there exists a copy of v_i, on some supertask allocated to some processor which sends its data to p_k. When recalculating the start time of v_j, this particular copy of v_i must be considered.

As a matter of notation, let v_{v_i,p_l} be the copy of task v_i allocated to processor p_l, where $p_l \in processors(v_i)$. Let $\mathcal{I}(c)$ be as defined in Definition 5.4.1 (page 105). Thus, if task v_j is in the cluster c, and there exists an immediate predecessor v_i of v_j, such that $v_i \in \mathcal{I}(c)$, then there exists a supertask $c' \in V_s$ which sends v_i's data to $proc(c)$. Let $p_l = proc(c')$ and $p_k = proc(c)$. Therefore, $v_{v_i,p_l} \in pred(v_{v_j,p_k})$.

Each processor in $\mathcal{P} = \{p_0, \ldots, p_{m-1}\}$ has a list of tasks to execute. The ordering of execution of the supertasks, and consequently, of the tasks, stated by the schedule S_s is maintained. Let $Ord(v_{v_i,p_l})$ be the order of execution of the copy of tasks v_i on processor p_l. For each processor p_l, $Idle(p_l)$ denotes the time in which processor p_l is idle again, during the recalculation process.

Clearly, a task cannot start execution before receiving all the necessary data from its immediate predecessor. The earliest time that a task v_{v_j,p_k} can start execution is then

$$est(v_{v_j,p_k}) \;=\; \max_{v_{v_i,p_l} \in pred(v_{v_j,p_k})} \{f(v_{v_i,p_l}) + \mathcal{C}\tau\}$$

where $f(v_{v_i,p_l})$ is the finish execution time of tasks v_i on processor p_l and $\mathcal{C} = 1$ if $p_l \neq p_k$, and $\mathcal{C} = 0$, otherwise. All data sent to v_{v_j,p_k} may be available before processor p_k is ready to receive, i.e. while p_k is executing some other task. Therefore, the start time of v_{v_j,p_k} is then

$$s(v_{v_j,p_k}, p_k) \;=\; \max\{est(v_{v_j,p_k}), Idle(p_k)\}$$

The start time of v_{v_j,p_k} is recalculated only after the start time of the immediate predecessors in $pred(v_{v_j,p_k})$ have been re-established.

Procedure: 15 $unbundled\_messages(G_s, S_s)$ $\{$

determine all tasks $v_{v_i,p}$ and their relationship;
$\forall p \in \mathcal{P},\; Idle(p) := 0 \wedge next(p) := 1;$
$\forall v_{v_i,p} \in V_s$ $\{$
    $D(v_{v_i,p}) := |\, pred(v_{v_i,p})\,|;$
    if $((D(v_{v_i,p}) = 0) \wedge (Ord(v_{v_i,p}) = next(p)))$ $\{$
        $A := A \cup \{v_{v_i,p}\};$
        $s(v_{v_i,p}, p) := 0;$
    $\}$
$\}$
while $(A \neq \emptyset)$ $\{$
    $v_{v_j,p}$:= element of A;
    $A := A - \{v_{v_i,p}\};$
    $est(v_{v_j,p}) := \max_{v_{v_i,p'} \in pred(v_{v_j,p})}\{f(v_{v_i,p'}, p') + \mathcal{C}\tau\};$
    $s(v_{v_j,p}, p) := \max\{est(v_{v_j,p}), Idle(p)\};$
    $Idle(p) := s(v_{v_j,p}, p) + 1;$
    $next(p) := next(p) + 1;$
    $\forall v_{v_t,p'} \in succ(v_{v_j,p})$
        $D(v_{v_t,p'}) := D(v_{v_t,p'}) - 1;$
    $\forall v_{v_i,p'} : D(v_{v_i,p'}) = 0 \wedge next(p') = Ord(v_{v_j,p_k})$
        $A := A \cup \{v_{v_i,p'}\};$
$\}$

$\}$

7.1.2.3 Latency and overhead are not negligible

The sending and receiving overheads may have small values if compared with the latency τ, but these may be non-negligible values. In this case, if during each iteration i the superedges defined are associated with low weights (relative to the current sets $\Delta'(c)$ and $\mathcal{I}'(c)$) due to the application of the *Minimal Subset Choice* Rule, the resulting schedule may have a makespan far from the optimal. The application of this rule may produce many communication events that may be incurred when the supertasks communicate. Also, applying the *Maximal Subset Choice* Rule may not provide a good schedule, due to the significant value of τ. In the attempt to overcome this problem, another parameterisation of the *Subset Choice* Rule is proposed with the objective to balance the weights among the superedges incident to each supertask c.

Suppose that during an iteration i, the set $\Delta'(c)$ contains the candidate clusters c_k and c_j. These two candidates can be related in the following way:

1. $(c_k \cap \mathcal{I}'(c)) \subset (c_j \cap \mathcal{I}'(c))$. Let c_j be the candidate cluster during iteration i which has the maximum-cardinality subset of $\mathcal{I}'(c)$. We give preference to c_j as the immediate predecessor. Therefore, c_j is selected as immediate predecessor of c and c_k is not included in $\Delta'(c)$ during subsequent iterations, due to the updating of the set $\mathcal{I}'(c)$ at each iteration;

2. $(c_k \cap \mathcal{I}'(c)) \cap (c_j \cap \mathcal{I}'(c)) \neq \emptyset$ and $(c_k \cap \mathcal{I}'(c)) \not\subset (c_j \cap \mathcal{I}'(c))$, and $(c_j \cap \mathcal{I}'(c)) \not\subset (c_k \cap \mathcal{I}'(c))$. Let c_j be a candidate with the maximal-cardinality subset of $\mathcal{I}'(c)$ during the current iteration i and suppose that $\mid c_k \cap \mathcal{I}'(c) \mid < \mid c_j \cap \mathcal{I}'(c) \mid$. Then, c_j is selected as an immediate predecessor of c and c_k is included on the set $\Delta'(c)$ defined in the subsequent iteration. The selected candidate c_j sends the largest message, removing the need for c_k to send part of this data. The candidate c_k is selected during some subsequent iteration $l > i$ only if $c_k \cap \mathcal{I}'(c)$ is the subset of the $\mathcal{I}'(c)$ established during iteration l with maximal-cardinality. On the other hand, if there is no such iteration l, c_k it is never selected as an immediate predecessor of c;

3. $(c_k \cap \mathcal{I}'(c)) \cap (c_j \cap \mathcal{I}'(c)) = \emptyset$. If the candidates c_j and c_k are selected during iterations i and $i+1$, respectively, the data sent by c_j does not affect the amount of data sent by c_k.

In the second situation enumerated above, both c_k and c_j have common data items so that they could "share" the responsibility of sending these data items, so that the weights associated with the superedges could be balanced. The idea of a rule which has the objective to balance the weights among the superedges incident to each supertask c is described next. We shall refer as the *Balanced Subset Choice* Rule.

7.1.2.4 The *Balanced Subset Choice* Rule

Let the $\Delta'(c)$ be the restricted set defined during iteration i. The *Balanced Subset Choice* Rule establishes that candidate clusters to be maintained in $\Delta'(c)$ are those which correspond to the *maximal-cardinality subset of the set* $\mathcal{I}(c)$. It must be borne in mind the difference between $\mathcal{I}(c)$ and $\mathcal{I}'(c)$. The former, is a fixed structure, in which elements are not removed during $phase_c$. On the other hand, the set $\mathcal{I}'(c)$, initialised as $\mathcal{I}(c)$, is updated at each iteration, through the removal of the tasks which belong to the selected candidate during that iteration. Therefore, $\mathcal{I}'(c)$ is a variable structure. Let the number of immediate predecessors in $\mathcal{I}(c)$ found in each cluster $c' \in \Delta'(c)$, be referred as $ntasks_b(c', c)$, and formally defined as:

$$ntasks_b(c', c) \ = \mid c' \cap \mathcal{I}(c) \mid$$

Then, let

$$x_b = \max_{\forall c' \in \Delta'(c)} \{ntasks_b(c', c)\}$$

The clusters kept in the set $\Delta'(c)$ are those containing x_b tasks from $\mathcal{I}(c)$. Let X_b be a subset of candidate clusters, each of which has less than x_b tasks from $\mathcal{I}(c)$, i.e.

$$X_b = \{c' \in \Delta'(c) \mid ntasks_b(c', c) \neq x_b\}$$

then, this subset of candidates is eliminated from $\Delta'(c)$:

$$\Delta'(c) = \Delta'(c) - X_b$$

In the same manner, if the resulting $\Delta'(c)$ has a single element, an immediate predecessor of c is found and must be included in the list $Predecessors(c)$. This inclusion is followed by the updating of $\mathcal{I}'(c)$. Otherwise, if $\mid \Delta'(c) \mid > 1$, further rules are applied to this set. Note that iterations are carried out while the set $\mathcal{I}'(c)$ is not empty.

7.1.2.5 Consequences of the *Balanced Subset Choice* Rule

Let $\Delta'(c) = \{c_k, c_j, \ldots, c_l\}$ be the restricted set established during iteration i and used as the list of candidates to be evaluated by the *Balanced Subset Choice* Rule.

Consider the tasks c_k and c_j already mentioned in the previous section, where $(c_k \cap \mathcal{I}'(c)) \cap (c_j \cap \mathcal{I}'(c)) \neq \emptyset$ and $(c_k \cap \mathcal{I}'(c)) \not\subseteq (c_j \cap \mathcal{I}'(c))$ and $(c_j \cap \mathcal{I}'(c)) \not\subseteq (c_k \cap \mathcal{I}'(c))$. Suppose, w.l.o.g., that

$$| c_j \cap \mathcal{I}(c) | \geq | c_k \cap \mathcal{I}(c) | \geq | c_l \cap \mathcal{I}(c) |$$

Thus, c_j is the candidate with the maximal-cardinality subset of $\mathcal{I}(c)$. The candidate c_j is included in the list $Predecessors(c)$ and the subset of tasks in $c_j \cap \mathcal{I}'(c)$ are removed from $\mathcal{I}'(c)$.

During iteration $i + 1$, the next selected candidate is c_k. Suppose that:

1. $(c_k \cap \mathcal{I}'(c)) \supseteq (c_l \cap \mathcal{I}'(c))$. In this case, c_l is not included in the restricted set $\Delta'(c)$ during the next iteration $i + 2$, since the set $\mathcal{I}'(c)$ is updated and used to construct $\Delta'(c)$ at every iteration;

2. $(c_k \cap \mathcal{I}'(c)) \subseteq (c_l \cap \mathcal{I}'(c))$, or $((c_k \cap \mathcal{I}'(c)) \cap (c_l \cap \mathcal{I}'(c)) \neq \emptyset \ \wedge \ (c_k \cap \mathcal{I}'(c)) \not\subseteq (c_l \cap \mathcal{I}'(c)))$. In this case, c_l is included in the restricted set $\Delta'(c)$ during the next iteration $i + 2$. It must be pointed out here that if the *Maximal Subset Choice* Rule was applied, c_k would not be even a candidate cluster during iteration $i + 1$, because c_l contains a larger subset of $\mathcal{I}'(c)$ than c_k. With the application of the *Maximal Subset Choice* Rule, the sending of data items to c may not be shared between c_j and c_k.

When $\mathcal{I}'(c)$ is empty, it is assigned again to $\mathcal{I}(c)$ and the add_edge procedure is performed using the ordered list $\mathcal{L}$ associated with $Predecessors(c)$. Bear in mind that the *Balanced Subset Choice* Rule is a proposal to be used in order to provide a better superdag in case that the overhead parameters are not negligible but they are smaller than the latency τ. Candidate clusters selected by the application of the *Balanced Subset Choice* Rule may have common tasks with $\mathcal{I}(c)$ and therefore, the burden of sending data items may be shared among them. Also, when effectively defining the superedges through the add_edges procedure, the smallest packets are defined before the largest ones.

Effects produced by the application of the *Balanced Subset Choice* Rule need still to be analysed. These effects are related with the out degree and in degree of the resulting supertasks. For example, let $c_j = \langle v_1, v_2, v_3, v_5, v_6, v_8 \rangle$, $c_k = \langle v_1, v_2, v_3, v_4, v_9, v_{10} \rangle$ and $c_l = \langle v_4, v_5, v_7, v_{11}, v_{12}, v_{13} \rangle$ be the candidate clusters in $\Delta'(c)$. Suppose that $\mathcal{I}(c) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$. Initially, $\mathcal{I}'(c) = \mathcal{I}(c)$. By applying the *Balanced Subset Choice* Rule, the first candidate selected is c_j, since $c_j \cap \mathcal{I}(c)$ is the subset of $\mathcal{I}(c)$ with maximal-cardinality. The set $\mathcal{I}'(c)$ is updated to $\{v_4, v_7\}$ and $Predecessors(c) = \{c_j\}$. Another iteration is initiated, when the candidate c_k is selected due to the fact that it contains the maximal-cardinality subset of $\mathcal{I}(c)$. Thus, $Predecessors(c)$ becomes equal to $\{c_j, c_k\}$ and the set $\mathcal{I}'(c)$ is updated to $\{v_7\}$. Finally, the candidate c_l is selected and $phase_c$ finishes because $\mathcal{I}'(c) = \emptyset$. Just before defining the respective superedges, $\mathcal{I}'(c)$ is again assigned $\mathcal{I}(c)$ and the superedges are specified in the following order:

1. c_l has the minimal-cardinality subset of $\mathcal{I}'(c)$, sending a total of 3 data items correspondent to the tasks v_4, v_5, v_7. $\mathcal{I}'(c)$ is updated to $\{v_1, v_2, v_3, v_6\}$;

2. (c_k, c) is now defined and the data resulted from the tasks v_1, v_2, v_3 is sent through this superedge. The resulting $\mathcal{I}'(c)$ is now equal to $\{v_6\}$;

3. Finally, the superedge (c_j, c) is defined, through which the data resulted from the task v_6 is sent. $\mathcal{I}'(c)$ becomes empty and current phase achieves its end.

The selected candidates may have a reasonable amount of common tasks, since they are compared with the set $\mathcal{I}(c)$ which remains the same throughout $phase_c$ (note that they do not send the same data). These selected candidates may be visited at future phases. For example the supertasks c_j and c_k have as common tasks the ones in the set $\{v_1, v_2, v_3\}$. When c_k and c_j are visited, as the immediate predecessors of these common tasks are not allocated neither to c_k-cluster nor to c_j-cluster, c_k and c_j may have equivalent sets of candidate clusters, i.e. $\mathcal{I}(c_j) \simeq \mathcal{I}(c_k)$. Consequently, a higher priority is given to the candidate clusters in $\Delta'(c_j)$ and $\Delta'(c_k)$ which have more common tasks with $\mathcal{I}(c_j)$ and $\mathcal{I}(c_k)$, respectively. As a result, the out degree of the immediate predecessors of supertasks c_k and c_j may have a higher out degree than in the case in which the sets of candidate clusters of distinct supertasks are disjoint.

If the *Maximal Subset Choice* Rule is applied to the example described above, executing the original $superdag$ algorithm provides the following definition of superedges:

1. the superedge (c_j, c) is specified, through which the data items produced by tasks v_1, v_2, v_3, v_5, v_6 are sent;

2. the superedge (c_l, c) is specified which sends the data originated from the tasks v_4 and v_7.

Concluding, the use of the *Balanced Subset Choice* Rule is just a proposal. An evaluation of its practicability and the range of the communication parameter values need yet to be investigated.

7.1.3 Degree

The rule related to the degree is associated with the *sending events*. Consider a model in which the sending overhead is significant. In this case, the out degree of the supertasks, mainly those in the critical path of the superdag, must be minimised. In such a case, the selection of the candidate with the smallest out degree does not necessarily affect the size of the message sent by the selected supertask. If the *Degree* Rule is performed after the selection of the message, it does not influence the definition of the weight of the resulting superdags. When λ_s has a significant value, and either $\lambda_s \geq \tau$ or $\lambda_s \leq \tau$, it is crucial to maintain a low out degree of the supertasks when defining the superedges.

On the other hand, if λ_s is negligible such that there is no cost associated with the *sending* events and therefore, there is no need to perform this rule. As already discussed before, τ is the only parameter considered, it is better to select many immediate predecessor supertasks sending small messages rather than a few immediate predecessors sending long messages. Consequently, the algorithm should allow the definition as many superedges as necessary. Due to the replication of tasks, the definition of small messages may lead to the formation of supertasks with higher out degree than when defining large messages.

7.1.4 Same interval

Section 5.4.4.6 describes the reason for the selection of candidate clusters which are scheduled at latest interval. The copies of any task u_i are replicated during two adjacent intervals (Lemma 5.3.3). Suppose that $u_i \in \mathcal{I}(c)$, for some visited cluster c. Note that, the decisions are dependent on the values of γ and λ_s. However, bear in mind that we are considering that the overheads are critically high. Suppose, w.l.o.g., that u_i belongs to the two candidate clusters c' and c^*, as shown in Figure 7.1, where c is scheduled during band $\beta' - 1$. When this Rule is performed during $phase_c$, all of the candidates have the same out degree, say d. If the cluster at the earliest interval c' is selected, its out degree becomes $d + 1$. The problem arises when for another visited cluster c^+ during band β', where $u_i \in \mathcal{I}(c^+)$, and therefore, c' is the only candidate that contains the task u_i (since there is no other candidate at earlier intervals which contains u_i). Note that the overhead λ_s can be reasonably high, even it may be the case that $\lambda_s > \gamma$ and each band is $\gamma + 1$ wide. Also, the increase of the out degree of c' may propagate when other clusters are visited during band $\beta' - 1$ and β'.This situation cannot be allowed to occur.

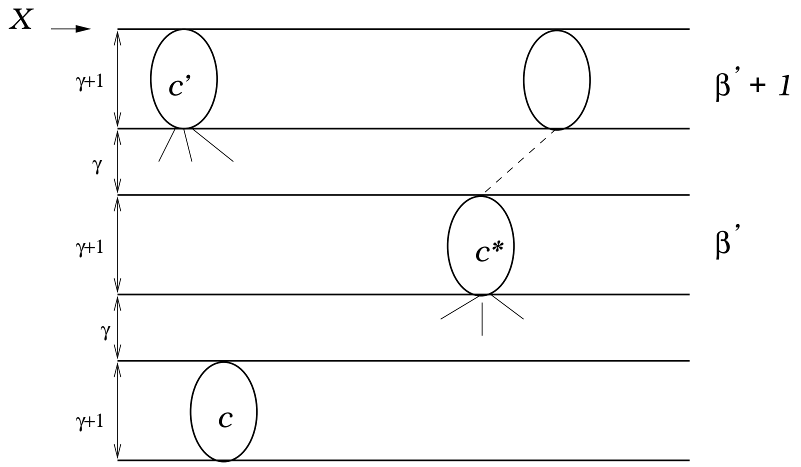


Figure 7.1: Selection of the latest or earliest candidates depending on the communication parameters.

Therefore, the *latest* candidate cluster c^* is selected in order to break ties at the current $phase_c$.

On the other hand, if the overheads are negligible and the delay is the only significant parameter, the out degree of the supertasks does not hinder the effectiveness of the resulting schedule. In the example shown in Figure 7.1, the selection of c' as the immediate predecessor of c allows that the message (c', c) is transmitted while supertasks (there exists at least one supertask at each band) are executed at band β'. Therefore, in the *delay* model, this rule can be parameterised as the *Earliest Interval* Rule.

Note that both *Degree* and *Same Interval* Rules are mechanism implemented mainly to control the out degree of the supertasks.

7.1.5 Smallest topological order

When a candidate c' is select through the *Smallest Topological Order* Rule as immediate predecessor of a cluster c, the c'-owner is the furthest task from c-owner, amongst the candidates. This implies that $\mid c \cap c' \mid$ is low and (as a consequence) the number of common tasks between these two clusters is small.

Analysing the case in which this rule is applied to the original set $\mathcal{I}(c)$, the selection of candidates with the same smallest topological order corresponds to the fact that clusters with fewest common tasks in common with $\mathcal{I}(c)$ are being selected. Consequently, these selected candidates can send small messages.

As already discussed, when the *delay* model is considered, it is better to send many and small messages, since τ is dominant. Therefore, we propose to use the *Smallest Topological Order* Rule as the first rule executed after the restricted search space is defined. In this case, not only are small messages sent but also, the degree of replication is minimised and the immediate predecessor tasks will send their results as early as possible.

7.2 The environment of the experiments

In order to evaluate the benefits given by MSA, comparison with existing heuristics should be done. If we could find a heuristic which takes into account the same communication parameters, a fair comparison would be performed. The *Mapping Heuristic* (MH) proposed by El-Rewini *et al.* [41] was considered since it incorporates startup cost for sending a message. Note that MH incorporates mechanisms in order to minimise congestion. However, congestion is not implemented in our MH version (our version corresponds to the first MH proposed).

The results of MSA will be compared with the results of MH and also ETF. We know that ETF does not incorporate sending and receiving overheads. In order to be fair with ETF, the overheads were incorporated to the cost function $r(v_i, p_j)$ that guides ETF decisions. Also, after ETF has produced the schedule of an application, a prediction of the communication events is performed. These are the results compared with MSA.

Experiments will be carried out considered various values of the communication parameters and different models. Since MH only incorporates the overhead due to the sending of messages, this heuristic will only be used when $\lambda_r = 0$.

Rather than comparing the *predictions* produced by the various heuristics, i.e. the makespan of the schedules produced for different number of processors, we wish to compare the real performance of the schedules on a *real machine*. Nevertheless, with the aim to have an environment that can represent different parallel architecture, a *simulated machine* was implemented. The *simulator* is used as a platform for our experiments.

In the following subsections MH and the *simulated machine* are described.

7.2.1 The Mapping Heuristic

MH is a list scheduling algorithm which takes into account the *level* of tasks in order to schedule tasks onto physical processors. The architectural parameters of the target machine which are considered are described as follows. The set of processing elements is $\mathcal{P}$, where $m = \mid \mathcal{P} \mid$ is the number of physical processors. Although heterogeneous processors may be considered, we only define homogeneous. For adjacent processors, τ is the latency associated with their communications. The function $h(p, p')$ represents the number of hops between processors p and p'; the time to initiate the sending of a message by a processors is denoted by λ_s.

Characteristics of the application are captured by a *DAG* $G = (V, E)$. The execution cost of task v_i is represented by $m(v_i)$ and $w((v_i, v_j))$ is the amount of data sent from task v_i to v_j. The total communication delay between tasks v_i and v_j when executed on distinct processors p_k and p_l, respectively, is denoted by $C(v_i, v_j, p_k, p_l)$. Therefore, the total communication delay due to communication of tasks v_i and v_j allocated to processors p_k and p_l is

$$C(v_i, v_j, p_k, p_l) = (w((v_i, v_j)) * \tau + \lambda_s) * h(p_k, p_l)$$

This communication cost function must be minimised in order to the minimum makespan be achieved.

7.2.1.1 Outlines of MH

Like ETF [65], MH is a list scheduling algorithm and in each iteration, it evaluates the start time of an *available* task on *idle* processors. Unlike ETF, MH has no notion of *current moment*, but it calculates the start time of a free task on all processors of the network, considering the time that each processor is *ready* to executed it. *Ready time* of a processor p_i is the time when p_i has finished to execute an assigned task and is ready to execute a new one. Among each task-processor pair evaluated, the processor in which an *available* task starts earlier is selected. The *level* of a task v_i, denoted by $Level(v_i)$ is the priority associated with the tasks for its selection and is defined as the length of the longest path from the sink task to v_i, considering the communication costs associated with the edges.

MH is guided by the occurrence of *events* which leads the algorithm to a certain "state". There are two *states* in which the heuristic can be met: "there is an available task" state, denoted by $\mathcal{F}$ and "there is a scheduled task" state, denoted by $\mathcal{S}$. When the algorithm initiates, the source tasks of the *DAG* are naturally free to be scheduled and therefore, the state $\mathcal{F}$ is generated. After a tasks is scheduled, the state $\mathcal{S}$ is generated, allowing the immediate successors of the just scheduled task to be examined, checking whether they are free or not. As long as the states are generated, they are inserted in a *queue* denoted $State$. The heuristic is executed by selecting the first element of $State$ and performing the correspondent operation, as seen in Algorithm 16.

Algorithm: 16 $mapping\_heuristic()$ {

```
read the DAG G = (V, E);
set parameters of the architecture;
∀v ∈ V,  calculate Level(v);
∀v ∈ V do
    if (succ(v) = ∅)  State := State + {F};
while (State ≠ ∅) {
    s := first(State);
    if (s = F){
        schedule the free task v with highest Level(v);
        State := State + {S};
    } else if (s = S){
        Let v be a task just scheduled;
        if (∃u ∈ succ(v) | all predecessors are already scheduled )
            State := State + {F};
    }
}
}
```

7.2.1.2 Improvements on the algorithm

Adam *et al.* [1] evaluated different priorities used in list scheduling algorithms and concluded that the priorities that should be used when choosing the next *available* task to be scheduled are the highest level, the *available* task with the largest number of immediate successors and the random selection. El-Rewini *et al.* implemented these priorities in MH, so that ties are broken, in the following order: 1) the *available* task with the highest level; 2) the *available* task which starts earlier; 3) the *available* task with the largest number of immediate successors; 4) select any *available* task at random. The procedure $best\_task\_processor()$ incorporates such priorities, as shown in Algorithm 17.

Algorithm: 17 $best\_task\_processor()$ {

```
start := ∞;
Let L be the list of free tasks v with highest Level(v);
```

```
    ∀v ∈ L {
        ∀p ∈ 𝒫 {
            ready := Ready_Time(p);
            if (pred(v) = ∅)
                earliest := ready;
            else{
                msgt := max_{u∈pred(v)}(f(u) + (w((u, v)) * τ + λ_s) * h(proc(u), p));
                if (msgt > ready)
                    earliest := msgt;
                else
                    earliest := ready;

                if (start > earliest){
                    start := earliest;
                    task := v;
                    proc := p;
                }
            }
        }
    }
}
```

7.2.2 Brief description of the simulator

In order to analyse the predictions produced by various heuristics, an environment that simulates a "real machine" is implemented. The importance of the use of such simulator instead of a real environment is due to mainly two factor: at first, architectural parameters may have different values so that different "machines" are being simulated and secondly, it would be difficult to perform the resulting schedules on a real machine having many factors of a real environment can affect the time to completion of the schedule performance.

The information given to the simulated machine comprises of the application, represented by a *DAG* or superdag; the execution costs of the tasks (supertasks), the weight associated to each edge (superedge) and the schedule produced by a given heuristic. Regarding the architectural characteristics implemented in the simulator, the following parameters are defined:

- the delay associated with communication of the smallest message that can be transmitted from one processor to an adjacent processor;
- the overheads associated with the sending and receiving events executed by a processor;
- cost associated with the switching of context of the execution of a task whenever a receiving event occurs in a processor;
- the topology of the network.

A processor is able to compute a given task, to send messages and to receive messages. A task is delivered to the processor which is specified in the given schedule. After the processor finishes to execute the task, it sends all messages required by the immediate successors of the task, so that they can be executed. Thus, after the execution of a task, *sending event* tasks are executed by the processor.

When a message is sent from a source processor p_i to a destination processor p_j and p_j is executing a task t_k, the message is not kept in a queue waiting for p_j to finish the execution of t_k. Instead, the tasks t_k is preempt and the processor p_j prepares itself to receive the message and an overhead associated with the *receiving event* may be incurred. Immediately after the receive event, the execution of t_k continues. If there is no other request for receiving messages, the processor continues in the execution mode, otherwise, the preemption incurs again. There may be a time spent during the preemption of the task and when its execution is restarted: it is the *time-for-preemption*, which is a constant.

Queues are associated with each processor, in which requests are delivered. Whenever a processor attains a request, another request is fetch by it. However, a priority is given to the *receiving* requests. The processor checks if there is any request in the queue. If there are requests, the processor first try to select a *receiving* request, otherwise, it selects the first request in the queue. In networks other than fully connected, a processor can forward a message depending on whether the processor is the destination or not. The cost of receiving and sending this message may be paid to the processor.

Note that throughout this dissertation, the cost *time-for-preemption* is considered negligible and the topology of the network is fully connected.

7.3 Concluding remarks

The different orders and parameterisation of the rules that guides the *multi-stage approach* is a way to adapt MSA so that it can give good schedules for different target models. Basically, we are going to consider two different models: the *delay* model and the CLAUD model. The definition of the overhead parameters in CLAUD is the characteristic the differentiates this model from the *delay* model.

We wish to focus the evaluation of the performance of MSA, mainly in the model in which the overheads are much higher than the delay (actually, usually $\lambda_s, \lambda_r \gg \tau$). Also, we wish to verify MSA versatility in models like the *delay* model. Cases that are not realistic are also considered for the sake of evaluation of the performance of MSA. These cases are also part of the set of experiments and analysis.

Chapter 8

Experimental Analysis

This chapter is dedicated to experimental evaluation of MSA and the analysis of different orders and parameterisation of the *rules* implemented in the *superdag* algorithm. At first, experiments are done on a specific instance *DAG* which is a irregular *DAG*. Different target models are considered as an opportunity to analyse the impact of the rules specified in MSA.

Secondly, a series of random graphs generated similarly as in [1], are given to MSA. The results are analysed and compared with ETF and MH. Note that since MH only considers the delay on communication and overhead when sending messages, this heuristic is only used in the experiments in which $\lambda_r = 0$.

8.1 Irregular input *DAG*

An irregular *DAG* which represents the molecular physics code [75] is used as input to a series of experiments, where the results of MSA are compared with other scheduling heuristics. We know that one application may not be a general example and may not capture many characteristics that are relevant when analysing a given strategy. However, with this example we are able to evaluate more carefully the performance of our proposed strategy, discussing and observing different points in the strategy.

As specified in the $superdag$ algorithm, the rules applied to the clusters produced by the *replication* algorithm are listed in the order in given Section 5.4.4. In all of the cases described below, the same rules are applied to the input *DAG* in Figure 8.1, but the two Rules *Subset Choice* and *Topological Order* are performed in different orders and parameterisation. Note that the *Degree* Rule may be ignored, depending on the target model considered. The difference between the **Tests** are related with the architectural parameters.

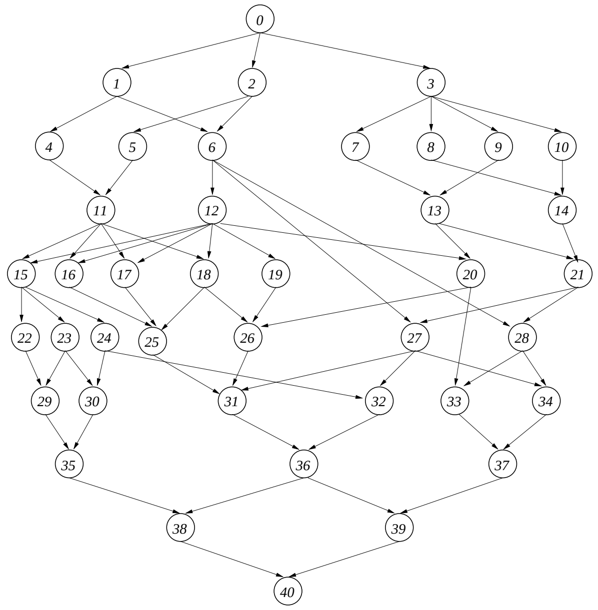


Figure 8.1: An irregular *DAG*: Molecular Physics Code.

Note that the rules are executed during the first stage of MSA at the $superdag$ algorithm, in order to construct the input superdag for the *remapping* algorithm performed at the second stage.

8.1.1 The delay as the only parameter

Test 1: In this first case, the only significant parameter is considered to be the delay τ. The parameters λ_s and λ_r are negligible and assume value zero. The rules performed in the $superdag$ algorithm are:

1) Candidate Clusters Set;
2) Restricting the Set;
3) Unique Copy;
4) *Smallest Topological Order*;
5) *Minimal Subset Choice*: small messages are sent;
6) Same Interval; (note that the *Degree* Rule is not performed)

The resulting superdag is given to the *remapping* algorithm which produces the final schedule. Since there is no overhead due to the sending and receiving of messages, the *unbundled-messages* procedure produces the final schedule. The results are compared with the scheduling results provided by the ETF and MH algorithms.

Two experiments considering $\tau = 2$ and $\tau = 6$ are carried out, correspondent to a low and a high communication costs, respectively. Note that the prediction and time to completion of the simulation for both MSA and ETF are the same. In both experiments, the prediction of MSA, i.e. the makespan of the schedules produced by MSA, is practically the lowest, producing better results for larger number of processors. As the value of τ is increased, the results of ETF get worse and MSA is better. The results produced by the algorithm MH and the simulation times are also given. Not only the makespan of MH schedules are the highest one, but also, the prediction is not very accurate, although it is an upper bound on the time to completion of the simulation. The results are shown in Figures 8.2 and 8.3. Recall that, since the overhead parameters are negligible, the schedule in which messages are *unbundled* is given to the simulated machine and thus, computation can totally overlap with communication. In Figure 8.3, where a high delay is considered ($\tau = 6$), we observe the bad performance of heuristics like ETF and MH. There is no improvement on the makespan when the number of processors is more than one. Actually, when $m > 1$, ETF provides schedules with higher makespan than that when $m = 1$.

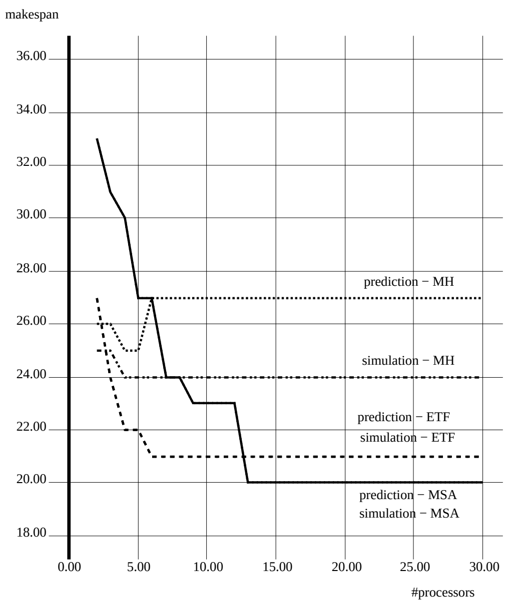


Figure 8.2: Comparing ETF, MH and *unbundled-messages* of MSA and their simulations for $\tau = 2$.

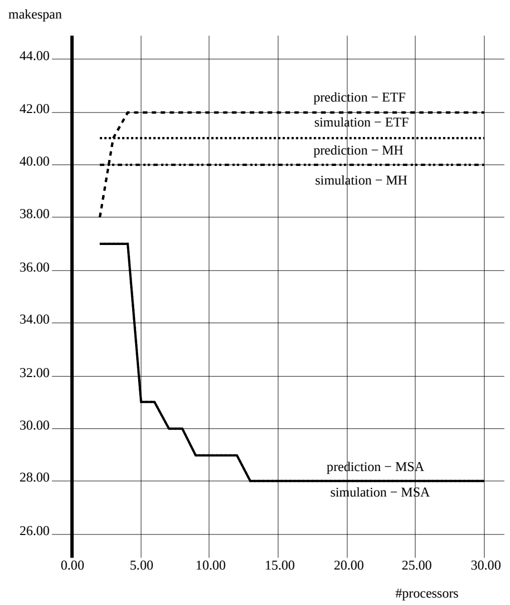


Figure 8.3: Comparing ETF, MH and *unbundled-messages* of MSA and their simulations for $\tau = 6$.

Test 2: The objective of this second test is to compare the schedule produced by MSA in **Test 1** and the schedule produced by MSA when the order of the rules are:

1) Candidate Clusters Set;

2) Restricting the Set;
3) Unique Copy;
4) *Minimal Subset Choice*;
5) Same Interval;
6) *Smallest Topological Order*;

As in **Test 1**, the delay τ is considered to be the only significant parameter. The predictions of the MSA are very close to those produced by **Test 1** for low values of τ. However, when τ has a higher value, the results provided by **Test 2** are considerably worse. Figures 8.4 and 8.5 show the comparison.

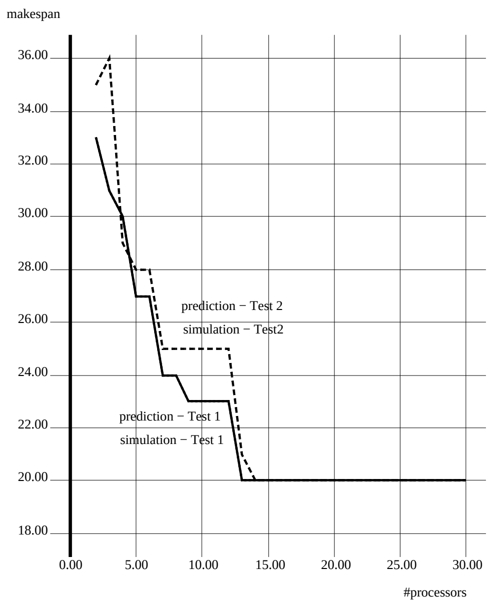


Figure 8.4: Comparing **Tests 1** and **2**, both produced by MSA for $\tau = 2$ and different rules order.

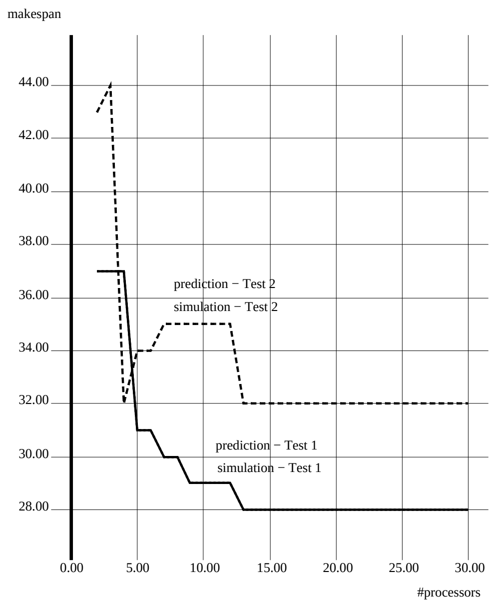


Figure 8.5: Comparing **Tests 1** and **2**, both produced by MSA, for $\tau = 6$ and different rules order.

The selection of immediate predecessors with smaller topological order (of their owners) leads to the definition of adjacent supertasks with a smaller set of common tasks between them. Consequently, a selected cluster also has a smaller number of tasks in $\mathcal{I}(c)$, for a given visited cluster c, considering all the current candidates in $\Delta'(c)$. Breaking ties by immediately performing the *Smallest Topological Order* Rule does not allow that other rules interfere with the selection. In **Test 2**, since this rule is the last one to be performed, the break of ties is done firstly by the *Same Interval* Rule, which can select clusters with higher topological order as supertasks, and therefore, longer messages may be specified since the candidates are closer. When unbundling these longer messages, more dependencies between tasks are defined, leading to a longer delay on the communications. Selecting the smallest topological order candidate as an immediate predecessor of supertasks in the superdag used as input to the *remapping* algorithm, may produce schedules with smaller makespan.

8.1.2 Overheads are predominant

Test 3: The sending overhead is now considered to be much higher than the others communication parameters. In this case, the *unbundled-messages* procedure is not performed, since the superdag concept is crucial for the effectiveness of the schedule. The purpose of this experiment is to highlight the effect of defining maximal-cardinality messages when the target machine has significant overheads. The order and parameterisation of the rules in the *superdag* algorithm are:

1) Candidate Clusters Set;
2) Restricting the Set;
3) Unique Copy;
4) *Maximal Subset Choice*: large messages are defined;
5) Degree ;
6) Same Interval;
7) *Smallest Topological Order*;

We refer to this order of rules as C_1. Another experiment, referred as C_2, is performed for the same values of the architectural parameters. The order of the rules is the following one:

1) Candidate Clusters Set;
2) Restricting the Set;
3) Unique Copy;
4) *Smallest Topological Order*;
5) *Maximal Subset Choice*;
6) Degree ;
7) Same Interval;

For both cases, schedules are produced and given to the simulated machine. The results are compared and shown in Figures 8.6 and 8.7. The first Figure considers the values of the architectural parameters as $\lambda_s = 4$, $\lambda_r = 0$ and $\tau = 0$. In the second Figure, the values are $\lambda_s = 4$, $\lambda_r = 1$ and $\tau = 0$. In both cases, it is observed that C_1 produces results with better time to completion than C_2. Let $\mathcal{M}(S_{C_i})$ be the results predicted by MSA when the order C_i is applied and

$\mathcal{E}(S_{C_i})$ is the correspondent time to completion of the simulation, $i = 1, 2$. In both cases, $\mathcal{M}(S_{C_1}) < \mathcal{M}(S_{C_2})$ and $\mathcal{E}(S_{C_1}) < \mathcal{E}(S_{C_2})$.

The situation in which the overheads are significant is different from the case in which the delay is the only significant parameter. When overheads are predominant, large messages must be sent, so that the $superdag$ algorithm produces supertasks with degree relatively low. This explains the results produced in C_2 where the priority is first given to those candidates with smallest topological order. This results in not very large messages and consequently, more superedges are necessary. On the other hand, when C_1 is applied, large messages are defined immediately which results in supertasks with smaller degree.

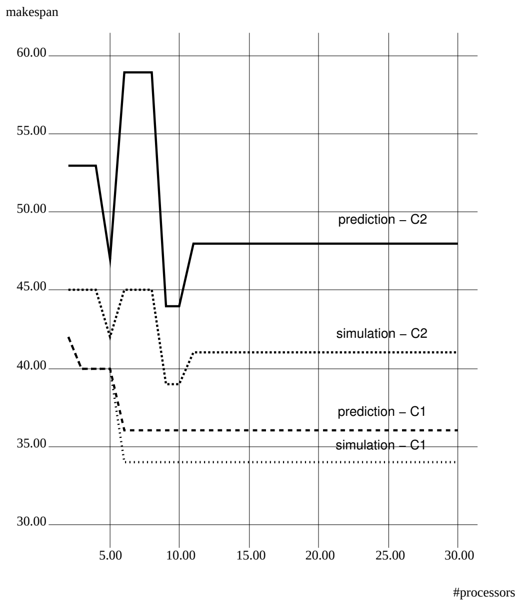


Figure 8.6: The parameters are: $\lambda_s = 4$, $\lambda_r = 0$ and $\tau = 0$.

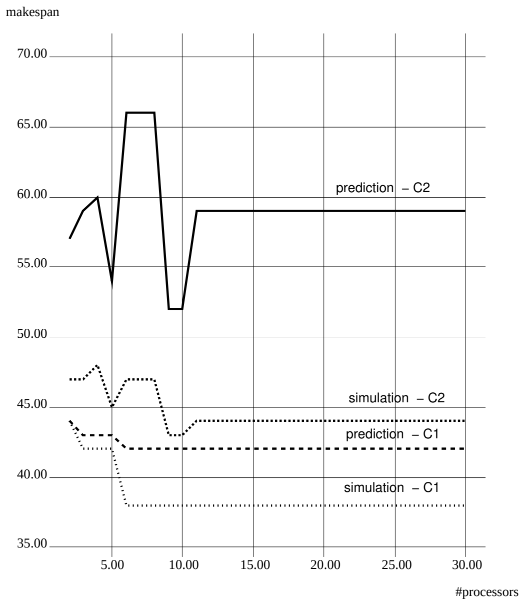


Figure 8.7: The parameters are: $\lambda_s = 4$, $\lambda_r = 1$ and $\tau = 0$.

Test 4: In this experiment, the results provided by the order and parameterisation proposed in experiment C_1 (significant sending overhead and a negligible delay) are compared with ETF and MH. The predictions and simulation results are shown in Figure 8.8 for $\lambda_s = 4$, $\lambda_r = 0$ and $\tau = 0$. The relation between the makespan provided by the three approaches is: $\mathcal{M}(S_{ETF}) < \mathcal{M}(S_{MH}) < \mathcal{M}(S_{MSA})$. However, the most important result is that the relation between actual execution time of the application when it is performed in the simulated machine, given each one of the schedules is $\mathcal{E}(S_{MH}) \gg \mathcal{E}(S_{ETF}) \gg \mathcal{E}(S_{MSA})$, given the schedule S_j.

Note that the MSA predictions are not exactly the same as the time to completion of the simulations, but the curves are closely correlated. The difference is practically equal to the value of λ_s. Nevertheless, MSA predictions can be considered an upper bound of the time to completion of the simulations.

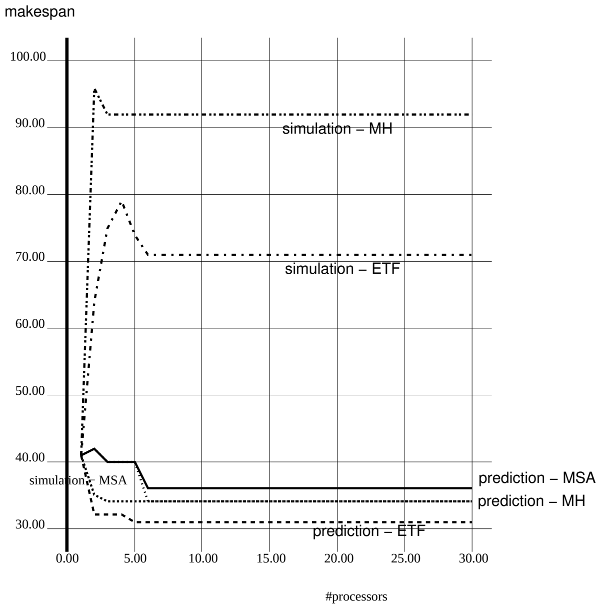


Figure 8.8: ETF, MH and MSA results for $\tau = \lambda_r = 0$ and $\lambda_s = 4$.

8.1.3 The *clustering factor*

On the investigation of the *clustering factor* γ used by MSA to produce its best schedule for a given application, the first stage of the algorithm is performed varying the values of γ. The irregular *DAG* $G = (V, E)$ in Figure 8.1 is the input application and for each value of γ, a specific superdag is produced. Let G^*_γ be the superdag produced by the first stage for the given γ and let γ_{best} be the *clustering factor* which defines a schedule with smaller makespan. Experimentally, the maximum number of parallel supertasks among all of the superdags produced is no more that 10 supertasks (for this specific *DAG* G) and in various experiments described below. This number is used as the number m given to second stage of MSA. The experiments carried out take into account different values for the communication parameters.

8.1.3.1 The *delay* model

The first set of experiments considers the *delay* model with delay $\tau = 2$, $\tau = 6$ and $\tau = 15$, such that low and high delays are analysed. The proper set of rules in the *superdag* algorithm is performed, as specified in **Test1** in Subsection 8.1.1. The *unbundled-messages* procedure is performed after the allocation of the supertasks onto physical processors is given by the *remapping* algorithm.

Also, since communication events costs are negligible, and the schedule of the communication event tasks are not defined in the final schedule, the predictions are very accurate. In Figure 8.9 each curve is associated with each value of τ for $\gamma = 0 \ldots, n-1$, where $n = \mid V \mid$. When $\tau = 2$, the communication cost is

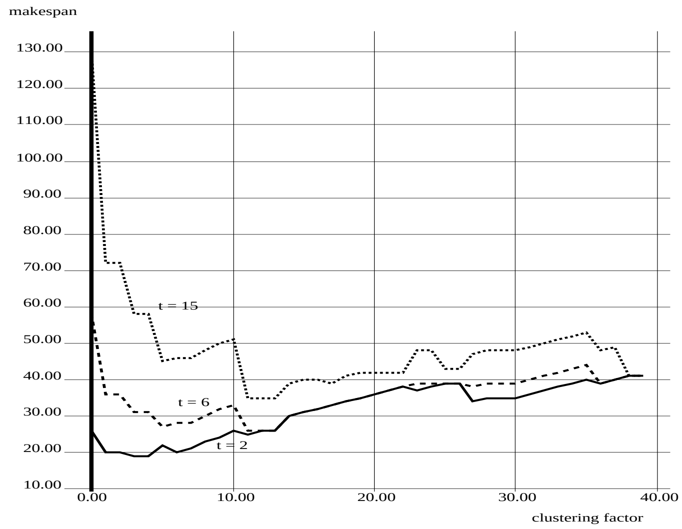


Figure 8.9: Makespan for the Schedules produced by MSA, considering the irregular *DAG*, $\lambda_s = \lambda_r = 0$, $m = 10$ and $0 \leq \gamma < 40$. The values for τ are 2, 6 and 15.

relatively low and the values for γ_{best} are both 3 and 4. Less computational and communication intervals are produced for high values of γ and also, the grain of the supertasks is large, thus decreasing the degree of parallelism. In this case, although the communication costs are relatively low, the resulting makespan is higher. On the other hand, as expected, for $\tau = 6$, γ_{best} is within the interval $[11, 14]$. The correspondent curve is flatter than the one correspondent to $\tau = 2$, because higher communication costs lead to the specification of schedules with makespan closer to the sequential one. When $\tau = 15$, γ_{best} is within the interval $[11, 13]$. The reason for the same optimal values for γ considering different communication costs is that, for the given G, when $\gamma > 10$, two computation intervals are produced.

Note that the γ_{best} is not always equal to τ. The tradeoff between the number of computational intervals and communication interval length is an issue to be given attention. Although the *delay* model is considered, where communication and computation may be totally overlapped, depending on the *structure of the input DAG*, communication intervals can be clearly defined, as in the case shown in Figure 8.10. Therefore, the structure of the *DAG* deeply influence the best value of γ.

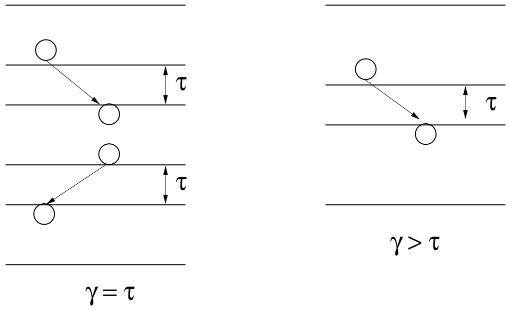


Figure 8.10: Tradeoff between the computational and communication interval length.

8.1.3.2 The *clustering factor* and the *sending event* costs

Consider that $\lambda_r = \tau = 0$ (negligible costs) and $\lambda_s \neq 0$. The cases in which $\lambda_s = 2$, $\lambda_s = 4$ and $\lambda_s = 8$ are evaluated. Figure 8.11 shows the results for $\lambda_s = 2$. We note that for small values of γ, MSA prediction is not very accurate. This is due to the fact that, the number of communication events may be relatively high. During the definition of the final schedule, it is difficult to accurately predict the order of events occurred during running time. The *update phase* of the *remapping* algorithm calculates the maximum number of *sendings* and *receivings* which are incurred during each communication interval. The final schedule represents an upper bound on the number of communication events.

As expected, for small values of γ the degree of parallelism is higher and good results may be produced when λ_s is not very large. When $\lambda_s = 2$, the best schedules were produced for $\gamma_{best} = 8, 12, 13$. When $\gamma_{best} = 8$, three computational intervals are specified and three *sending events* tasks affects the schedule. Regarding the schedule produced when $\gamma_{best} = 12$, two computational intervals are produced, which are longer than the one for $\gamma_{best} = 8$. However, only one *sending event* task affects the resulting schedule. The results are shown in Figures 8.11 and 8.12. For $\lambda_s = 8$, $\gamma_{best} = 12$, as verified in Figure 8.12, which is the same γ_{best} for $\lambda_s = 4$. The accuracy of the predictions reflects the fact that when the overhead λ_s is too high, the *sending event* tasks cannot overlap the actual transmission of messages as much as in the case of low overhead.

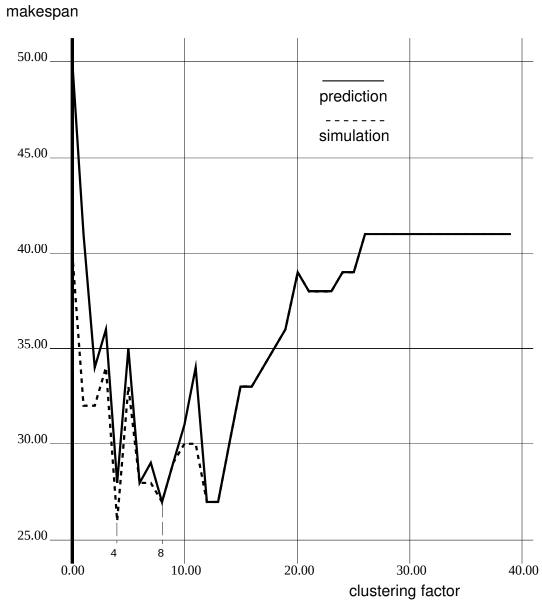


Figure 8.11: Communication costs: $\lambda_s = 2$ and $\tau = \lambda_r = 0$; $m = 10$ and $0 \leq \gamma < 40$

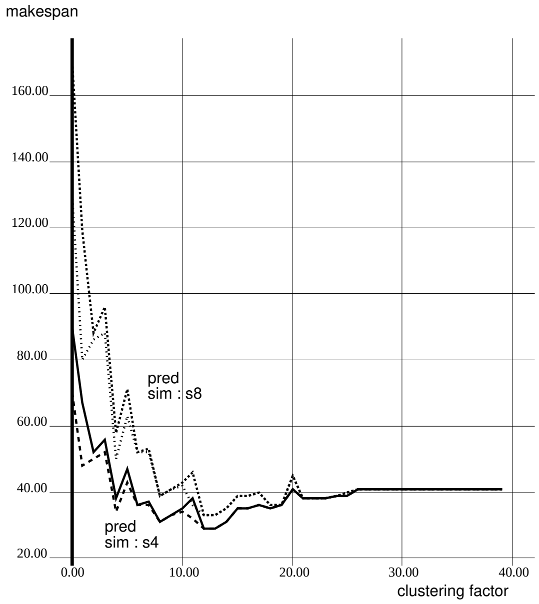


Figure 8.12: Communication costs: $\lambda_s = 4, 8$ and $\tau = \lambda_r = 0$; $m = 10$ and $0 \leq \gamma < 40$

8.1.3.3 The *clustering factor* and the *receiving event* costs

In the current experiment, $\lambda_s = \tau = 0$ and $\lambda_r = 2, 4, 6$ (again $m = 10$). In Figure 8.13 , the best schedule is produced when $\gamma_{best} = 8$ for all the three values of λ_r considered. It is not always the case that, increasing the *clustering factor* γ, the number of communication events is decreased, since the produced superdag is dependent on the precedence relation of G. For example, when $\gamma = 8$, three computational intervals are produced and two receiving events are incurred in each of the two communication intervals. When $\gamma = 10$, three computational intervals are also produced, but a total of *three receiving event* tasks affects the resulting schedule. Therefore, the increase of the makespan is characterised not only by the larger size of the supertasks in the latter case but also, by the higher number of communication events. Figure 8.14 shows the experiments for $\lambda_r = 4$ and $\lambda_r = 6$ (it is not worthy parallelising for $\lambda_r > 6$). An interesting point observed is that γ_{best} is the same for both values of λ_r values evaluated. This is due to the fact that when $\gamma_{best} = 8$, the number of *receiving* events which are incurred, is much smaller than for other values of γ. The curves in Figures 8.11 and 8.12 are flatter than the curves in Figure 8.13 and 8.14 due to the higher concurrency of *sending event* tasks and application tasks.

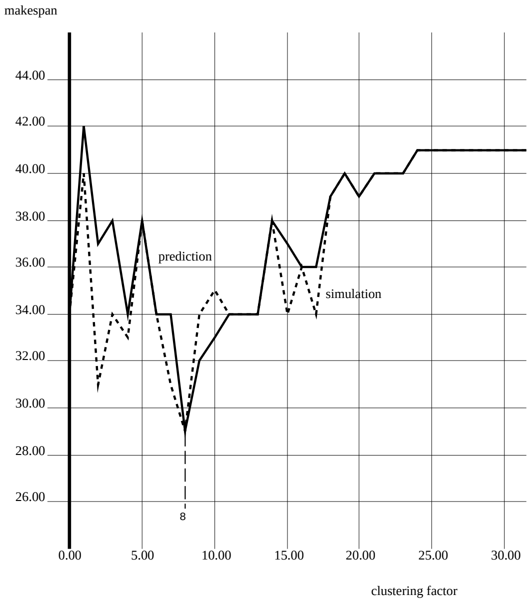


Figure 8.13: Communication costs: $\lambda_r = 2$, $\lambda_s = \tau = 0$; $m = 10$ and $0 \leq \gamma < 40$.

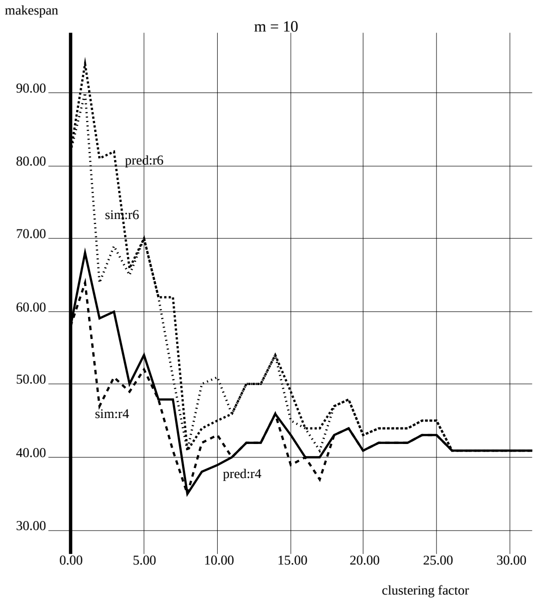


Figure 8.14: Communication costs: $\lambda_r = 4, 6$ and $\tau = \lambda_s = 0$; $m = 10$ and $0 \leq \gamma < 40$

8.1.3.4 Comparing the number of *sending* and *receiving events*

Figure 8.15 shows the curves representing the schedule produced when different values for γ is given to MSA, considering $m = 10$. One curve corresponds to the parameter values $\lambda_s = 2,\ \lambda_r = \tau = 0$ and the other curve corresponds to $\lambda_r = 2$,$\lambda_s = \tau = 0$. The high number of *sending events* can be verified when no replication is carried out, that is $\gamma = 0$. Providing enough processors, such as $m = \mathcal{W}(G_0^*)$, the resulting schedule $S(G_0^*)$ is such that $\mathcal{M}(S(G_0^*)) > \mathcal{M}(S_{seq})$, where S_{seq} is the schedule of G on one processor. Through the clustering of tasks, messages are bundled and the number of *sending events* is decreased. On the other hand, when the *receiving* overhead is the only significant parameter, the schedule without replication has smaller makespan than the sequential one for the considered instance. However, better results are achieved for higher values of γ. The comparison shows that the out degree of supertasks produced is relatively smaller than their in degree. In both cases, $\gamma_{best} = 8$.

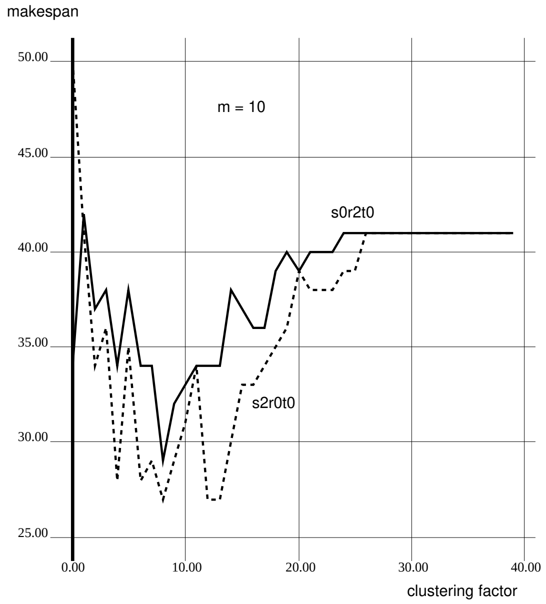


Figure 8.15: Comparing $\lambda_r = 2, \tau = \lambda_s = 0$ and $\lambda_s = 2, \tau = \lambda_r = 0$. For both cases, $m = 10$ and $0 \leq \gamma < 40$

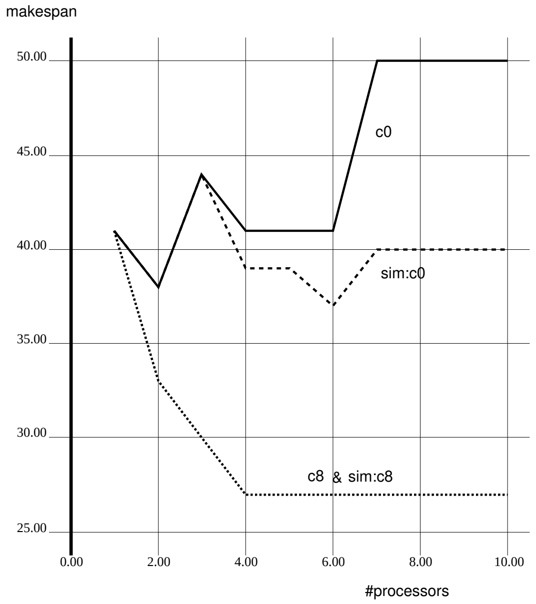


Figure 8.16: The curves for $\gamma = 0$ and $\gamma = 8$, for $\lambda_s = 2, \tau = \lambda_r = 0$ and varying m.

8.1.3.5 On the number of processors

The experiments in the previous subsections considered that $m = \mathcal{W}(G^*_\gamma)$, which does not necessarily mean that the smallest makespan is achieved for this value of m. Let $\lambda_s = 2$, $\lambda_r = \tau = 0$ and the values of γ be 0 and 8. Let the second stage be performed for the values of m within the interval $[1, 10]$. Two curves are produced, each one representing the allocation of G^*_0 and G^*_8 onto m processors, $m = 1, \ldots, 10$, as seen in Figure 8.16. For $\gamma = 0$, since there is no replication of tasks and consequently, no bundling of messages, the number of communication events may be reduced by allocating G^*_0 to a smaller number of processors and its best result is achieved when $m = 2$. Also, when the number of processors increases, the prediction of the schedule of G^*_0 is not accurate due to the large number of *sending events* which are incurred. The benefits the bundling of messages is verified when $\gamma = 8$. The makespan of the schedule of G^*_8 is smaller than that of G^*_0 for any value of $m > 1$.

8.2 Analysis on various irregular *DAG*s

Various random *DAG*s generated in a similar manner as the one specified in [1] are used as input *DAG*s to out experiments. Mainly, two cases are evaluated: one correspondent to the CLAUD model, where the overhead parameters are much greater than the delay (usually, $\lambda_s > \lambda_r \gg \tau$), and the other case corresponds to the *delay* model, in which the only significant parameter is τ. In Tables 8.1 and 8.2 the results produced by MSA and ETF are listed considered that the parameter values characterise a CLAUD machine. Table 8.3 shows the results produced by MH, only for those experiments in which $\lambda_r = 0$. Since the *sending overheads* are predominant and both λ_s and λ_r are greater than τ, the order and parameterisation of the rules of the *superdag* algorithm used to produce G^* is:

1) Candidate Clusters Set;
2) Restricting the Set;
3) Unique Copy;
4) Maximal Subset Choice;
5) Degree ;
6) Same Interval;
7) Smallest Topological Order;

Of course, in this case, no unbundling of messages is performed after the remapping onto m processors is established. Table 8.4 contains the results produced for the *delay* machine. In this case, the set of rules in the *superdag* algorithm are:

1) Candidate Clusters Set;
2) Restricting the Set;
3) Unique Copy;
4) Smallest Topological Order;
5) Minimal Subset Choice;
6) Same Interval;

In this case, after the second stage, the *unbundled-messages* procedure is performed. The schedules produced are given to the simulated machine (CLAUD or *delay* "machine", correspondingly).

Tables 8.1 and 8.2 comprises of the following information. In the first column the input *DAG* G is the random *DAG* R_n, where n is the number of tasks. Then, the following three columns represents the values of the architectural parameters λ_s, λ_r and τ. The next five columns are related to the results provided by MSA: γ is the best *clustering factor* used to produce the superdag G^* and $\mathcal{W}(G^*)$ is its width. The best value of γ is evaluated iteratively and corresponds to the value that produces the smallest makespan, given that the number of processors is greater than or equal to the width of G^*. Then, performing the second stage for G^*_γ considering a varying number of processors,

m is derived, which corresponds to the number of processors that produces the smallest makespan. The best makespan is denoted by $\mathcal{M}$. Sim is the time to completion of the respective simulation. The next four columns are related to the results provided by ETF, where m is the number of processors that produces the schedule with the smallest makespan, $\mathcal{M}$ is this makespan and Sim is the time to completion of the simulation. Note that the last column of Tables 8.1 and 8.2 presents the smallest time to completion of the simulation of a scheduled produced by ETF for the number of processors given between brackets. This column aims to show that it is not always the case that the best prediction of ETF corresponds to the best simulation results (when it does, — is shown).

High sending overheads: when λ_s is the only significant value, or, at least, the most significant one, MSA produces the best schedule and also the prediction is very accurate. Even when ETF produces a schedule with smaller makespan, as observed in some cases, the time to completion of the respective simulation is much higher than what was predicted. Practically, in all of the

Parameters				MSA					ETF			
G	λ_s	λ_r	τ	γ	$\mathcal{W}(G^*)$	m	$\mathcal{M}$	Sim	m	$\mathcal{M}$	Sim	best Sim †
R_{80}	4	0	0	13	27	27	43	43	10	17	108	67(19)
	0	4	0	21	25	7	69	73	1	80	80	60(27)
	4	4	0	21	25	1	80	80	1	80	80	—
R_{153}	4	0	0	19	59	59	64	64	8	33	185	128(33)
	0	4	0	40	41	41	120	125	36	104	78	63(57)
	4	4	0	50	32	32	132	132	47	125	181	153(1)
R_{108}	8	0	0	22	30	30	86	85	9	51	248	108(1)
	0	8	0	85	6	6	97	97	1	108	108	—
	4	8	0	85	6	6	97	97	1	108	108	—
	8	4	0	85	6	6	97	97	1	108	108	—
	4	4	0	85	6	6	97	97	1	108	108	—
	8	4	2	85	6	6	97	97	1	108	108	—
R_{170}	8	0	0	7	98	98	33	33	63	23	62	49(50)
	0	8	0	19	68	68	37	44	83	78	61	55(63)
	2	4	3	21	68	68	45	43	41	52	58	49(33)
	8	0	8	26	67	67	47	37	43	38	77	60(58)
R_{223}	8	0	0	8	112	112	37	36	35	29	82	39(106)
	8	0	8	33	77	77	54	38	18	46	120	55(113)
	0	8	8	40	74	74	73	42	91	127	110	—
R_{298}	15	0	0	16	133	133	104	104	55	55	241	211(104)
	0	15	0	137	56	56	155	140	75	250	219	171(90)
	15	4	8	137	56	56	155	140	34	165	357	240(97)
R_{256}	8	4	0	39	79	79	117	123	50	97	170	125(83)
	15	8	6	92	61	61	127	121	113	202	344	256(1)
	15	0	6	92	61	61	119	113	71	73	183	171(106)
R_{124}	4	2	0	7	60	60	26	23	14	29	42	29(48)
	2	4	0	13	49	49	28	27	20	37	49	36(37)
	4	2	4	7	30	60	31	25	26	35	42	34(42)

† the smallest simulation time for ETF

Table 8.1: The *claud* model.

cases in which $\lambda_s > \lambda_r \gg \tau$, the prediction of MSA is an upper bound of the respective simulation and both curves (prediction and simulation) are correlated. This can be expected since when performing the *updating* phase (in the *remap-*

ping algorithm), the upper bound on the number of communication events is calculated. Therefore, MSA is more accurate.

Parameters				MSA					ETF			
G	λ_s	λ_r	τ	γ	$\mathcal{W}(G^*)$	m	$\mathcal{M}$	Sim	m	$\mathcal{M}$	Sim	best Sim †
R_{286}	15	4	1	96	44	1	286	286	40	253	569	286(1)
	1	4	15	48	73	1	286	286	38	248	256	244(79)
	15	15	0	147	28	1	286	286	1	286	286	—
R_{310}	15	0	0	40	84	84	163	163	35	90	460	310(1)
	20	10	5	129	1	48	310	310	1	310	310	—
	30	10	0	129	1	48	310	310	1	310	310	—
R_{357}	8	4	0	61	100	100	119	119	163	93	124	115(94)
	10	4	1	61	100	100	129	121	156	104	164	134(94)
	15	4	4	107	96	96	142	125	85	138	243	186(76)
R_{364}	15	4	1	95	55	55	268	256	26	215	504	275(99)
	1	15	4	183	7	32	363	337	1	364	364	279(125)
R_{152}	8	0	0	22	59	59	109	109	23	48	284	152(1)
	15	8	0	93	3	20	136	134	1	152	152	—
	0	8	8	94	1	18	152	152	1	152	152	—
R_{186}	8	2	1	14	72	72	69	59	28	61	187	137(38)
	2	8	1	31	54	54	83	87	69	103	105	—
	15	8	1	32	26	25	95	91	30	153	340	186(1)
R_{234}	8	2	0	67	57	57	183	165	47	112	247	230(64)
	8	0	2	27	90	90	151	144	19	60	431	234(1)
	2	8	8	142	3	28	225	217	1	234	234	218(70)
	8	2	8	30	1	84	234	234	16	155	487	234(1)
R_{154}	4	2	1	17	56	56	77	59	32	49	88	70(45)
	8	4	2	28	52	52	118	80	34	69	130	117(56)
	4	2	6	12	68	68	101	94	28	92	87	75(42)
R_{154}	4	2	1	24	52	52	64	54	21	50	63	94(35)
	1	2	4	9	39	39	72	55	21	49	114	25(25)

† the smallest simulation time for ETF

Table 8.2: The *claud* model.

In Figure 8.17, the results of the prediction and simulation produced by MSA are compared, considering a random *DAG* with 298 tasks and the communication parameters $\lambda_s = 15$, $\lambda_r = 4$ and $\tau = 8$. The superdag used is the G^*_γ, where $\gamma = 137$, since $\mathcal{M}(S(G^*_\gamma))$ is the smallest makespan for m = $\mathcal{W}(G^*_\gamma)$. Note that MSA predictions are an upper bound of the time to completion of the respective simulation. In Figure 8.18, the comparison with ETF is shown. The experiment highlights the importance of the superdag definition, even when the number of physical processors is small. When there are few processors, MSA produces "longer" supertasks and less communication events are incurred. In the case of ETF, there is no attempt to minimise the number of communication events, but the makespan is minimised by scheduling tasks in such a way that communication overlap useful computation. On the specification of the final schedule provided by ETF, an attempt to calculate the cost of sending and receiving messages is implemented. However, it is difficult to predict and calculate precisely, since decisions are done in accordance to the local state of the scheduling process. When the resulting schedule is given to the CLAUD machine, the time to completion is much higher than the ETF predictions. Figure 8.17 shows the close correlation between the prediction of MSA and the time to completion of the respective simulation in the CLAUD machine.

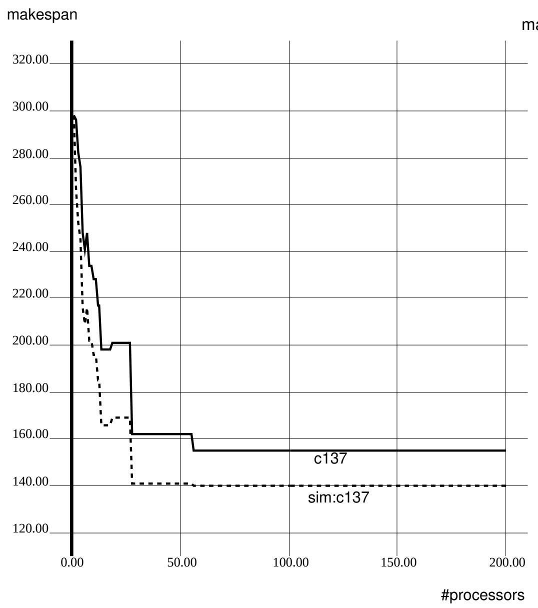


Figure 8.17: R_{298}, $\lambda_s = 15$,$\lambda_r = 4$ and $\tau = 8$. MSA best results for $\gamma = 137$.

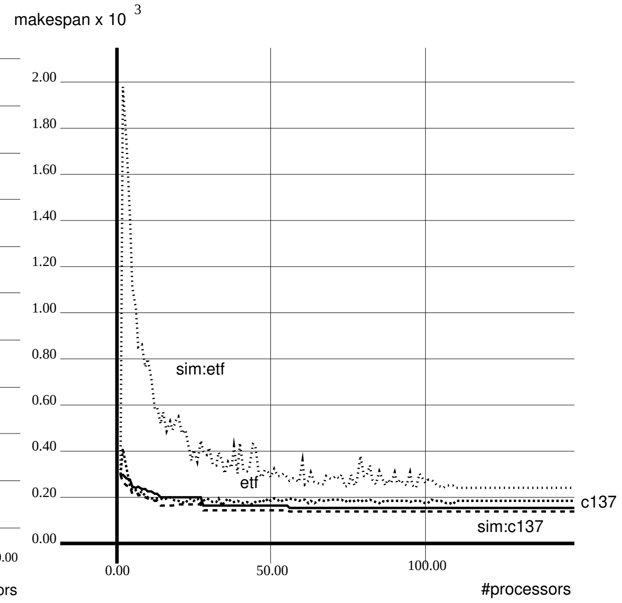
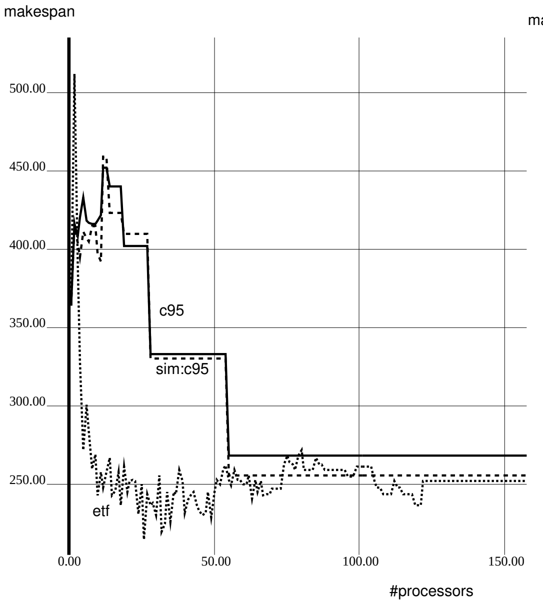


Figure 8.18: R_{298}, $\lambda_s = 15$,$\lambda_r = 4$ and $\tau = 8$. MSA results compared with ETF.

Figure 8.18 compares the MSA results with the ETF results. ETF predictions are very close to MSA results, however, the time to completion of the simulation of ETF schedules is very far from what was predicted, mainly when the number of processors is not very high.

Figures 8.19 and 8.20 show a similar situation for the random *DAG* with 364 tasks and the parameters $\lambda_s = 15$,$\lambda_r = 4$ and $\tau = 1$. In Figure 8.19, we note that the makespan of the schedules produced by ETF is smaller than MSA, for any number of physical processors. In the same Figure, we see how closely correlated are MSA predictions and the time to completion of the respective simulations. On the other hand, in Figure 8.20 the time to completion of the simulation is too far from ETF predictions.

As already stated, MH is only performed for those experiments in which $\lambda_r = 0$. In Table 8.3, predictions and time to completion of the respective simulation are shown. The experiments show that MH is totally inaccurate for a CLAUD machine. The time to completion of the simulations are too far from what was predicted. In all of the cases considered in Table 8.3, the respective MSA best results are very accurate. Therefore, MH does not predict the reality. Figures 8.21 and 8.22 show the total inaccuracy of the MH approach, when the sending overheads are significant, as well as the delay.



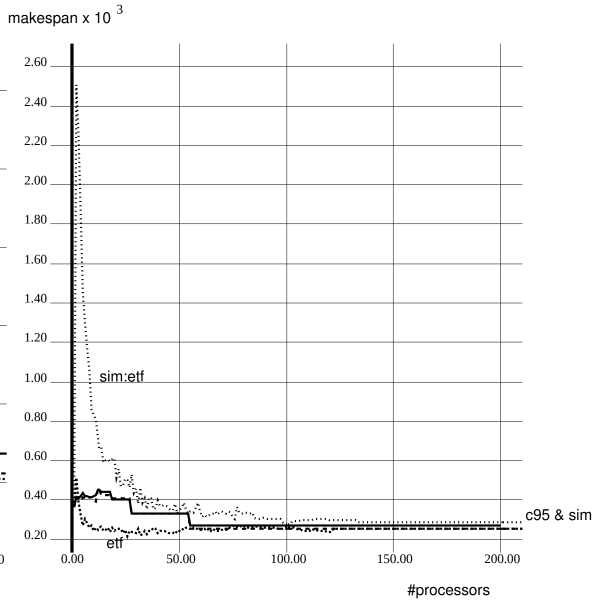


Figure 8.19: R_{364}, $\lambda_s = 15$,$\lambda_r = 4$ and $\tau = 1$. MSA best results for $\gamma = 95$ and the prediction of ETF.

Figure 8.20: R_{364}, $\lambda_s = 15$,$\lambda_r = 4$ and $\tau = 1$. MSA results compared with ETF.

Parameters				MH		
G	λ_s	λ_r	τ	m	$\mathcal{M}$	Sim
R_{80}	4	0	0	9	26	117
R_{153}	4	0	0	34	34	160
R_{108}	8	0	0	8	49	344
R_{170}	8	0	0	80	27	73
	8	0	8	41	42	73
R_{223}	8	0	0	94	24	84
	8	0	8	94	40	97
R_{298}	15	0	0	48	68	287
R_{256}	15	0	6	69	77	259
R_{310}	15	0	0	77	99	456
R_{152}	8	0	0	20	45	283
R_{234}	8	0	2	40	63	311

Table 8.3: MH results when λ_s and τ are significant.

The discrepancies between the ETF and MH prediction and their respective time to completion of the simulations is explained by the fact that these strategies minimise the makespan of their schedules by trying to overlap communication with computations. Therefore, there is no concern to minimise the number of communication edges, as far as computation of tasks is overlapped with communication. To aggravate the problem, this communication incorporates the startup overhead. These strategies do not minimise the number of communication events between distinct processors. When simulating the produced schedule, a processor the CLAUD machine executes each *communication event* tasks such that communication is not totally overlapped with computation of tasks in G. Therefore, the incorporation of overheads to the delay cannot be considered as a realistic approach.

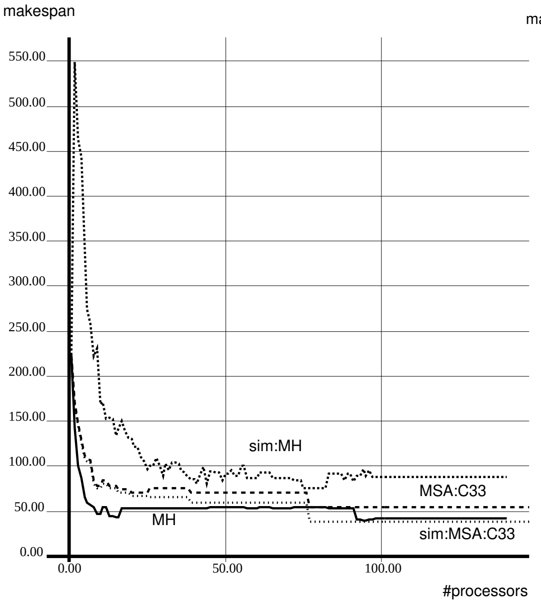


Figure 8.21: R_{223}, $\lambda_s = 8$, $\lambda_r = 0$ and $\tau = 8$. MSA best results for $\gamma = 33$ compared with MH.

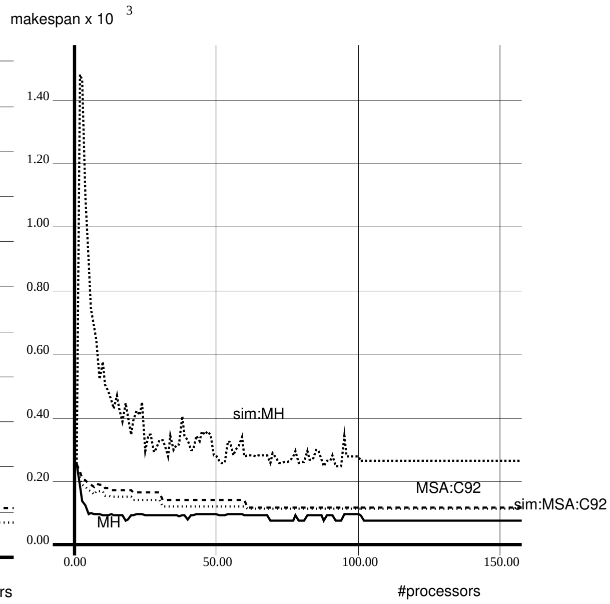


Figure 8.22: R_{256}, $\lambda_s = 15$, $\lambda_r = 0$, $\tau = 6$. MSA best results for $\gamma = 92$ compared with MH.

The receiving overheads: we also consider the case in which the value of λ_r is the only significant value. This is a hypothetical situation, since we cannot identify a parallel machine in which the *bottleneck* on the communication costs is the *receiving overhead* parameter. However, this case can help us to examine how MSA tackles the problem of minimising the number of *receiving events*. The results of MSA are not always the best. There are cases in which schedule produced by MSA has makespan slightly smaller than the time to completion of the simulation. Concluding, if, for some case, only the *receiving events* are the only relevant events, improvements in MSA should be performed.

The delay model: in the experiments considering the *delay* model, MSA produced better results than ETF and MH for all of the cases. The results of the experiments are shown in Table 8.4. In all of the cases experimentally evaluated, we note that $\frac{\tau}{2} \leq \gamma_{best} \leq 2\tau$. Again, the best value of γ is related to the structure of the *DAG* and to the delay τ. The best number of processors is exactly the width of the input superdag to the second stage, denoted by $\mathcal{W}(G^*)$. This is due to the either one of the facts: adjacent supertasks cannot be grouped together on the same processors in such way that interprocessor communication is zeroed, or the grain of the supertasks is large enough such that allocating them to the same processor increases the resulting makespan.

Parameters				MSA					ETF			MH		
G	λ_s	λ_r	τ	γ	$\mathcal{W}(G^*)$	m	$\mathcal{M}$	Sim	m	$\mathcal{M}$	Sim	m	M	Sim
ran1(140)	0	0	4	7	80	80	27	27	19	30	30	32	31	30
	0	0	15	18	63	63	49	49	13	70	70	14	87	87
ran2(80)	0	0	4	6	48	48	23	23	8	26	26	9	26	25
	0	0	20	20	39	39	38	38	8	59	59	7	71	71
ran3(153)	0	0	4	2	63	63	29	29	8	32	32	34	34	34
	0	0	15	8	87	87	48	48	14	73	73	9	87	87
ran4(108)	0	0	8	11	59	59	59	41	9	49	49	8	49	48
	0	0	15	11	60	60	54	54	10	66	66	13	78	78
ran5(170)	0	0	8	5	103	103	17	17	63	23	23	80	27	27
	0	0	15	20	77	77	23	23	43	36	36	41	40	40
ran7(298)	0	0	15	13	189	189	37	37	54	55	54	48	68	68
	0	0	20	13	189	189	42	42	54	70	70	48	88	88
ran9(124)	0	0	4	4	69	69	11	11	18	14	14	11	18	18
	0	0	15	4	69	69	21	21	26	35	35	10	44	44
ran10(286)	0	0	15	11	171	171	77	77	23	91	91	21	110	109
	0	0	20	13	167	167	92	92	35	114	114	21	136	136
ran12(310)	0	0	15	14	189	189	65	65	51	83	83	77	99	98
	0	0	20	14	195	195	75	75	18	102	102	77	124	124
ran14(364)	0	0	15	9	218	218	55	55	19	76	76	23	91	91
	0	0	20	9	218	218	65	65	19	96	96	23	116	116
ran15(152)	0	0	8	6	84	84	41	41	13	44	44	20	45	45
	0	0	15	17	88	88	50	65	24	70	70	20	73	73
ran16(186)	0	0	15	9	110	110	30	30	12	49	49	14	55	55
	0	0	4	3	95	95	18	18	15	22	22	15	22	22
ran18(234)	0	0	8	5	128	128	39	39	24	50	50	31	54	53
	0	0	20	11	145	145	81	81	26	96	96	40	113	113
ran19(154)	0	0	8	11	98	98	26	26	13	34	34	19	35	35
	0	0	15	11	98	98	33	33	11	59	59	19	54	54

Table 8.4: The *delay* model.

Although ETF and MH were scheduling heuristics proposed for the *delay* model, MSA gives better results, exploiting the degree of parallelism through Brent's Principle and minimising communication through replication. It must be pointed out that ETF, and even MH, may produce schedules with smaller makespan than those produced by MSA, considering a small number of processors and a relatively low delay, as seen in Figure 8.23. On the other hand, in machines with high delay (or even when the size of the messages is very large), MSA produces schedule with smaller makespan than both ETF and MH, even for a small number of processors. This is showed in Figures 8.24 and 8.25. In any case, if there are as many processors as the width of the correspondent superdag, MSA is a better approach for the *delay* model.

The evaluations show that MSA is very promising. It produces good schedules for MPP machines and for small network of processors, such as a network with few workstations which may be associated with high communication costs.

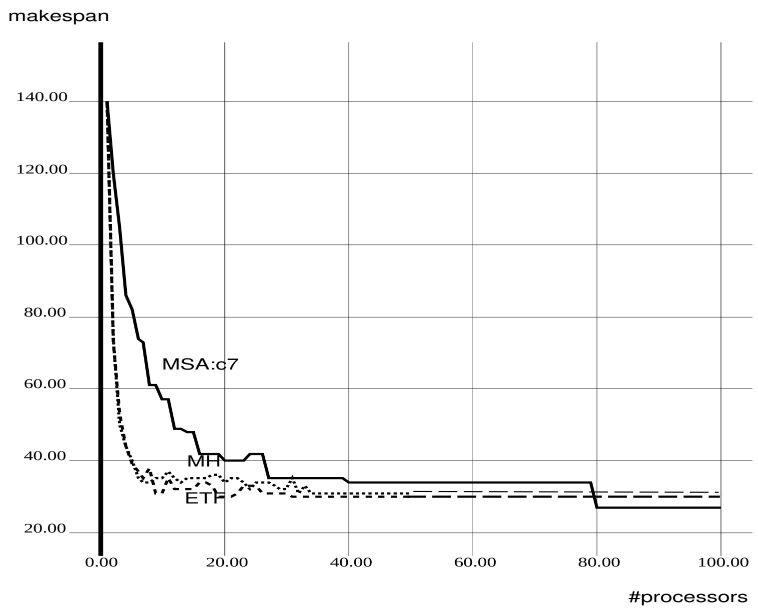


Figure 8.23: R_{140}, $\lambda_s = \lambda_r = 0$ and $\tau = 4$. Results of MSA for $\gamma_{best} = 7$, ETF and MH.

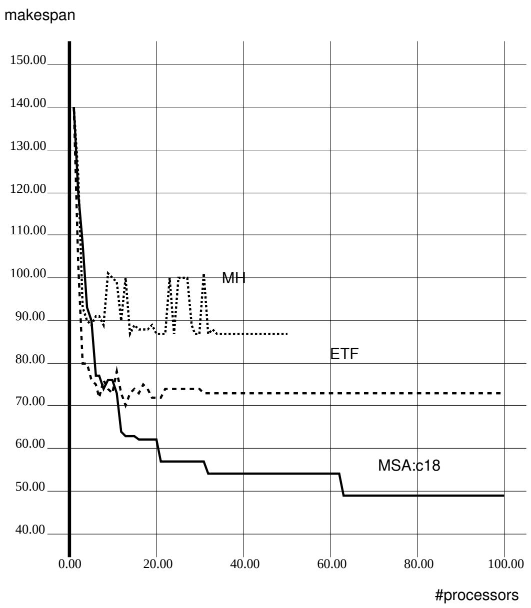


Figure 8.24: R_{140}, $\lambda_s = \lambda_r = 0$, long delay $\tau = 15$. MSA bes results for $\gamma = 18$ compared with ETF and MH.

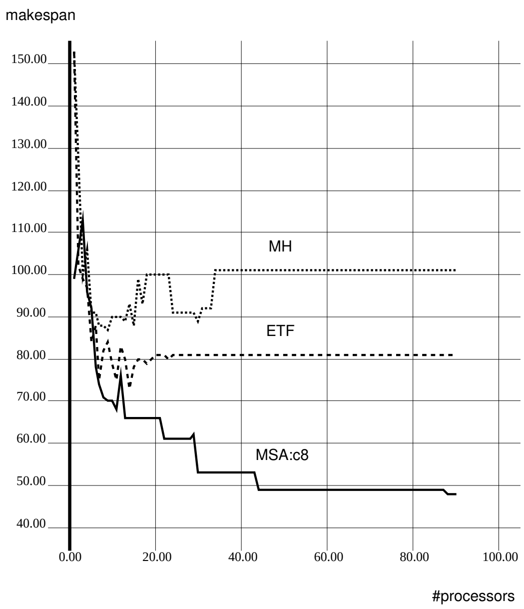


Figure 8.25: R_{153}, $\lambda_s = \lambda_r = 0$, long delay $\tau = 15$. MSA best results for $\gamma = 8$ compared with ETF and MH.

8.3 The Scheduler and Timings

The complexity analysis of each stage of MSA has already been discussed in Chapters 5 and 6. However in practical terms, a flavour of the actual execution time of the scheduler may be of interest and importance. Measurements of the

Input				MSA				Others	
DAG	m	$\lambda_s : \lambda_r : \tau$	γ	*replication*	*superdag*	*remap*	Total	ETF	MH
R_{140}	27	4:0:0	13	155187	1692109	315936	2163232	984473	62424
R_{140}	25	4:4:0	21	172752	709202	273595	1155549	1723221	58409
R_{298}	133	15:0:0	16	184482	987662	383412	1555556	28859037	378397
R_{298}	56	15:4:8	137	250831	139719	301292	691842	17277589	186569
R_{256}	79	8:4:0	39	161164	994465	300456	1456085	14526961	201251
R_{256}	61	15:8:6	92	172792	144709	285933	603434	13708500	163314
R_{310}	84	15:0:0	40	601085	1902014	361635	2864734	15232659	275216
R_{310}	48	30:10:0	129	941826	2449713	363602	3755141	13262741	177745
R_{234}	90	8:0:2	27	392845	3013562	386593	3793000	13197660	277590
R_{234}	57	8:2:0	67	481738	3041852	338311	3861901	9626735	185840
R_{154}	52	4:2:1	24	68314	204167	272470	544951	3397989	88190

Table 8.5: Strategy Timings in microseconds

time that MSA takes to produce schedules for a group of random *DAG*s on a CLAUD machine are presented in Table 8.5 together with the execution times of both the ETF and MH scheduling algorithms. It must be pointed out that the experiments were carried out on a multiuser system (Sun SPARCstation 20 with 192 Mbytes). The results shown were obtain by executing each experiment at least 60 times throughout the day during the processor's low load periods.

For a proposed methodology, MSA seems to be promising when comparing its execution time with ETF. On the other hand, MH is much faster. It must be pointed out that the current version of MSA has still to be tuned for performance, taking into consideration the details of implementation and optimisations on the structures that have been used. The objective of these experiments is to show the feasibility of MSA and its use. Note that MSA is proposed as a multi-stage strategy and therefore, a longer execution time for the scheduler would be expected.

8.4 Remarks

From the evaluation of the irregular *DAG* considering different communication parameter values, some conclusions can be derived regarding the *clustering factor* γ. The value of γ given to the first stage must be such that the number of communication events which are incurred are low and at the same time the degree of parallelisation is maximised. There is no *cost function* explicitly defined in the *replication* algorithm to guide the degree of replication (the amount of tasks clustered together). Decisions are made upon the evaluation of the structure of the *DAG*. Also, the rules defined in the *superdag* algorithm are under the assumptions of the values of the communication parameters.

Many scheduling algorithms proposed in the literature, like list scheduling heuristics define a cost function as a guide to schedule the tasks of the input

DAG. However, in general these heuristics lead to a local minima rather a global one, since there is no global view of the state of the system. When dealing with communication events as tasks, it is difficult to know the specific *clustering factor* that leads to the lowest number of communication events which are incurred. Therefore, the iterative approach is the best currently available, so that MSA can produced good schedules for a given *DAG*.

We considered cases in which each of the communication parameters were significant. Not necessarily all the cases represent realistic architectures. However, the evaluation also aims to verify how our strategy tackles on each communication parameter. For example, it is not realistic to assume that an architecture has the *receiving* overhead as the only communication cost. This case was considered for the sake of MSA performance analysis on the number of *receiving events* specification.

As we have seen, MSA is effective. Its main purpose is to tackle the problem of scheduling when the overheads are very high when compared with the delay, as in the case of many parallel machines. Also, MSA predictions are very accurate. We are also able to show the versatility of our approach in the experiments in which *delay* is the only significant parameter.

Chapter 9

On the Analysis of Regular Structures

9.1 Introduction

We now evaluate the schedule of certain regular *DAG*s, considering also different values for the delay and overhead parameters, their impact of the communication parameters when scheduling such *DAG*s. These special *DAG*s are: the trees and the *diamond DAG*. On the evaluation of the Full Binary Tree, we characterise two types of partitions and we shall see that depending on the target model, a certain partition produces the best schedule. Our *multi-stage approach* is applied and its versatility is verified, by tuning the input parameters that guide the clustering of tasks and *bundling* of messages, so that a suitable schedule for the underlying model is produced. The same analysis is done for the Inverted Binary tree and the diamond *DAG*. In this latter, known partitions are implemented considering the *claud* model. The respective results are compared with MSA on an attempt to evaluate its solution.

9.2 Fully Binary Tree

The Full Binary Tree $B_n = (V_B, E_B)$ is a *DAG* with n tasks, in which the direction of the edges in E_B are towards the unique sink in V_B. Each vertex, apart from the sink vertex, has one immediate successor and two immediate predecessors (apart from the source vertices). The characteristics of B_n are: there exists $\frac{n+1}{2}$ source tasks; the height of the tree is $l = \log(n+1)$; the tree is divided in levels, where the sources are at level $l-1$ and the sink at level 0; and each level i contains 2^i tasks, $0 \leq i < l$.

We shall see two partitions on the Full Binary tree and their impact considering the *delay* model, where the communication cost is solely characterised by τ, and also, the *claud* model, where the λ_s and λ_r overheads are also defined as communication parameters together with τ. For simplicity, in this section, whenever the overheads for sending and receiving are referenced as λ, it means that $\lambda_s = \lambda_r = \lambda$.

9.2.1 The *layer* partition

Based on the asymptotically optimal schedule provided by Papadimitriou and Yannakakis in [106], Jung *et al.* [70] devised a schedule strategy for the fully binary tree with n tasks taking into account the delay τ when communicating a data item between two neighbour processors. Jung *et al.* also provided the asymptotic lower bound

$$L = \tau \frac{\log(n+1)}{\log(\tau+1)} - \tau$$

for the schedule of the binary tree. The Papadimitriou and Yannakakis' strategy described in [106] for the binary tree matches the lower bound L.

The case addressed here incorporates another characteristic of architecture: the startup overhead λ. It must be borne in mind that the startup overhead is a blocking parameter, unlikely the *non-blocking* delay parameter τ modelled in [106] and [70], where each processor *multicasts* its results to various processors. Figure 9.1 shows the impact caused by these two communication parameters when scheduling a small *DAG* illustrated in Figure 9.1(a). The schedule in Figure 9.1(b) considers λ as the only communication parameter while the schedule showed in Figure 9.1(c), τ is the only parameter and communication can be totally overlapped with computation of tasks. In order to analyse the

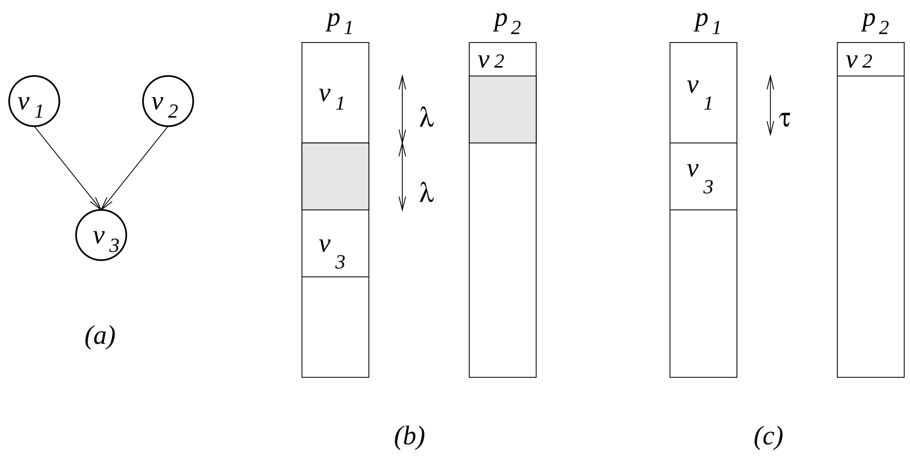


Figure 9.1: The full Binary Tree with 3 tasks: (b) Gantt Chart when $\lambda \gg \tau$; (c) when $\tau \gg \lambda$.

impact of the two parameters which characterise communication when partitioning the binary tree into layers, two models are considered: the *delay* model (s described in [106]) and the *claud* model.

First of all, we define the *layer* partition on the Full Binary Tree, abstracting from the knowledge of the communication parameters. We make use of the *clustering factor* parameter γ to define the layers. We shall see that, depending on the target model, γ may assume different values, i.e. it may be assigned to λ, τ or even some characteristic of the input *DAG*.

The binary tree B_n is partitioned into

$$h = \frac{\log(n+1)}{\log(\gamma+2)} = \log_{\gamma+2}(n+1)$$

layers, producing subtrees each comprising $(\gamma+1)$ tasks. Each subtree is $\log(\gamma+1)$ high and the layers are enumerated from $h-1$, in which the source tasks are located, to layer 0, which contains the sink task. The number of subtrees at each layer i is

$$a_i = (\gamma+2)^i$$

Note that a superdag is defined, in which each of the $\frac{n}{\gamma+1}$ supertasks is a subtree with $\gamma+1$ tasks from B_n. In order to provide simplicity in this analysis, the values chosen for γ are such that its possible to define a complete binary tree in each subtree. The in-degree of each supertask (apart from the sources) is equal to $\gamma+2$ and therefore, the resulting superdag is a $(\gamma+2)$-*ary tree*. The in-degree can be easily derived by observing that at layer i there exists, $a_i = (\gamma+2)^i$ subtrees which are equally connected to $a_{i-1} = (\gamma+2)^{(i-1)}$ subtrees at layer $(i-1)$. Note also that each superedge has a unit weight associated, since in each subtree, only the respective sink task sends data to immediate successor supertask. Figure 9.2 shows the layer partition on B_{63}, given that $\gamma = 2$. The circles indicate the supertasks of the resulting 4-ary tree and the edges coincide with the superedges.

9.2.1.1 The *layer* partition and the *delay* model

When scheduling a *DAG* $G = (V, E)$ considering the *delay* model, a delay τ is incurred when processors communicate. If $v, u \in V$ are tasks such that $(v, u) \in E$, u and v are executed on distinct processors so that task u is mapped to a processor p to which v has not been mapped, the task u can only start to be executed at least τ time units after the finishing execution time of a replication of task v. Jung *et al.* [70] proposed a layer strategy applied on the Binary

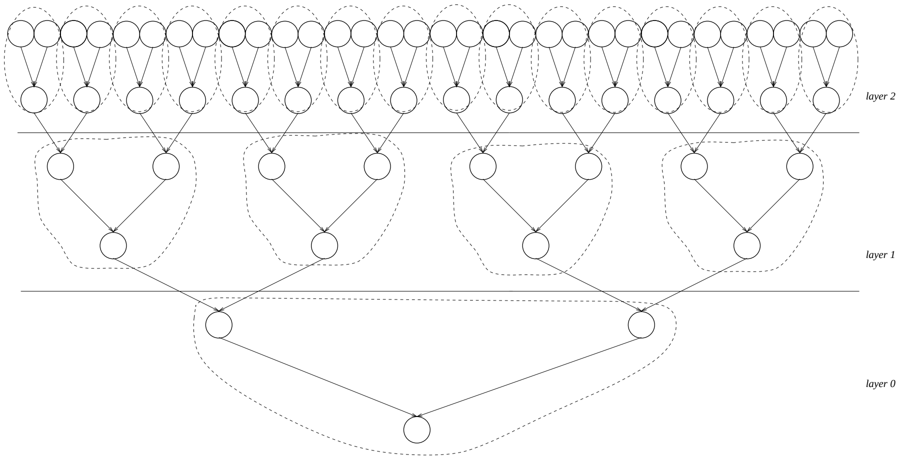


Figure 9.2: The Full Binary Tree with 63 tasks partitioned in layers, given $\gamma = 2$.

Tree B_n taking into account the *delay* model. Since the parameter τ guides the technique, the value τ is assigned to γ and the *layer* partition produces a $(\tau + 2)$-ary tree correspondent to a superdag.

Let the schedule S^{delay}_{layer} of the $(\tau + 2)$-ary tree provided by the *layer* partition under the *delay* model be defined as follows. Let c_i be a supertask allocated to a processor $p \in \mathcal{P}$. The elapsed time due to communication is equal to τ time units, no matters the number of immediate predecessors allocated to distinct processors, i.e. the communication time is not affected by the number of processors from which a given destination processor receives data. Nevertheless, if more than one immediate predecessor supertasks are grouped on the same processor, the computation time increases. Therefore, the minimum makespan of the schedule S^{delay}_{layer} provided by the layer partitioning of a binary tree B_n is achieved when the number of processors is equal to the number of source supertasks, i.e., $m_l = \frac{n+1}{\tau+2}$. The makespan of the schedule S^{delay}_{layer} of B_n onto m_l fully connected processors is then

$$\mathcal{M}(S^{delay}_{layer}) = h(\tau + 1) + (h - 1)\tau \tag{9.1}$$

9.2.1.2 The *layer* partition and the CLAUD model

The clustering of tasks onto processors is necessary when the amount of communication incurred when parallelising an application needs to be decreased, so that the time to completion is minimised. In the case of a binary tree, if each task is allocated to a distinct processor (no clustering is carried out), two communication events are incurred at each level along the path from each source to the sink. Depending on the communication parameter values and the number of tasks n, the amount of communication may be much greater than the computation time and therefore, clustering of tasks might be necessary. However, the question is how to cluster tasks considering the communication events in the way defined in the CLAUD model.

Let S_{layer}^{claud} be the schedule produced by the *layer* partition under the CLAUD model. Regarding the allocation, the supertasks at each layer are allocated to m processors. A supertask is allocated to the same processor as one of its immediate predecessor supertasks, so that the number of communication events is decreased. Note that the upper bound on the number of processors U_p is the number of supertasks in layer $h-1$, i.e. $U_p = (\gamma+2)^{h-1}$. The makespan of the schedule S_{layer}^{claud} is derived in the following Lemma.

Lemma 9.2.1 *The makespan of the schedule S_{layer}^{claud} of a Binary Tree B_n considering the* CLAUD *model onto a fully connected network with m processors is:*

$$\mathcal{M}(S_{layer}^{claud}) = (\gamma+1)\sum_{i=k}^{h-1}\frac{(\gamma+2)^i}{m} + k(\gamma+1) + (k-1)[(\gamma+2)\lambda+\tau] + \frac{m}{(\gamma+2)^{k-1}}\lambda + \frac{(\gamma+2)^k}{m}\tau$$

where $k = \lceil \log_{\gamma+2} m \rceil$ and $h = \log_{\gamma+2}(n+1)$ is the height of the tree.

Proof:
The number of processors m may be smaller than the number of supertasks a_i at some layer i of the $(\gamma+1)$-ary tree. Therefore, when partitioning a $(\gamma+2)$-ary tree onto m processors, the supertasks in each of the layers with more than m subtrees are divided evenly between the m processors. For the layers that contain less than or equal to m subtrees, obviously, less than or equal to m processors are used, each processor spending $(\gamma+1)$ time units to execute its supertask. Let $k = \lceil \log_{\gamma+2} m \rceil$, such that $m-1 \le a_k \le m$. Then, the total *computation time* produced by the *layer* partition is

$$\sum_{i=0}^{k-1}(\gamma+1) + (\gamma+1)\sum_{i=k}^{h-1}\frac{(\gamma+2)^i}{m} = (\gamma+1)\sum_{i=k}^{h-1}\frac{(\gamma+2)^i}{m} + k(\gamma+1)$$

$$= \frac{(\gamma+2)^h - (\gamma+2)^k}{m} + k(\gamma+1) \qquad (9.2)$$

For $0 < i < k$, each one of the processors executes one of the supertasks at layer i. These processors send concurrently one message to one of the $(\gamma+2)^{i-1}$ processors allocated to the supertasks at layer $i-1$. Each one of these destination processors, on the other hand, receives $(\gamma+1)$ messages, since one of its $(\gamma+2)$ immediate predecessor supertasks is allocated to the same processor, saving one sending and one receiving overheads which are incurred due to their precedence relationship. The transmission of the message takes τ unit time, since the weight of the superedges at these layers is unit. Therefore, the communication cost associated with the *sending* and *receiving* events which are incurred at layers 0 up to k and the transmission of the messages, is:

$$(k-1)[(\gamma+2)\lambda+\tau] \tag{9.3}$$

Unlike the binary tree in which the number of processors can be considered as a power of 2, the number of processors, allocated to a $(\gamma+2)$-ary tree, cannot be considered as a power of $(\gamma+2)$, since the value γ might be high. Therefore, a subset of the immediate predecessors of a supertask at layer $k-1$ might be allocated to a single processor. Consequently, the respective superedges are mapped onto a single link, which lays between the processor allocated to the subset at layer k and the processor allocated to the supertask at layer $k-1$. At layer k, there exists $(\gamma+2)^k$ supertasks, therefore, $(\gamma+2)^k$ concurrent superedges are mapped onto m links. The number of superedges mapped per link is

$$\frac{(\gamma+2)^k}{m}$$

Since $(\gamma+2)^k \geq m$, there exists m "new supertasks" (processors), each one being the result of the merge of $\frac{(\gamma+2)^k}{m}$ supertasks (subtrees). Therefore, the m new supertasks (processors) are connected to the $(\gamma+2)^{k-1}$ processors allocated to supertasks in the layer $k-1$. The total number of links through which communication is incurred with each one of these $(\gamma+2)^{k-1}$ processors at layer $k-1$ is

$$\frac{m}{(\gamma+2)^{k-1}}$$

Although one of the $(\gamma+2)^{k-1}$ processors at layer $k-1$ is the same as one of the m processors at layer k, it will be accounted for the concurrent *sending events*. Therefore, the total cost due to the communication events which are incurred between layers k and $k-1$ is:

$$\frac{m}{(\gamma+2)^{k-1}}\lambda \tag{9.4}$$

The number of supertasks allocated to a single processor at layer k corresponds to the number of sources of the respective subtrees which must send data items to the same immediate successor at layer $k-1$. Therefore, the total amount of data sent through one link is $\frac{(\gamma+2)^k}{m}$ and consequently, the delay on the communication is

$$\frac{(\gamma+2)^k}{m}\tau \tag{9.5}$$

Adding Equation 9.2 to the Equations 9.3, 9.4 and 9.5, the resulting makespan is

$$\mathcal{M}(S^{claud}_{layer}) = \frac{(\gamma+2)^h - (\gamma+2)^k}{m} + k(\gamma{+}1) + (k{-}1)[(\gamma{+}2)\lambda{+}\tau] + \frac{m}{(\gamma+2)^{k-1}}\lambda + \frac{(\gamma+2)^k}{m}\tau$$

$\square$

9.2.2 The *subtree* partition

There exist scheduling algorithms [16, 41, 46, 65, 75, 106, 116] which tend to allocate the critical path of a *DAG* to a single processor. When such algorithms are applied to the Binary Tree, one of the critical paths is allocated to a single processor. This allocation may be "inefficient" if the target machine has high startup costs associated when sending and receiving messages. Due to the structure of the Binary Tree, each task (apart from the source) can only start to be executed after its processors receive the data from the processors allocated to the immediate predecessors. Considering that only one of the immediate predecessors of a task is allocated to the same processor, each task can only start executing after a sending and a receiving event is incurred on the two respective distinct processors. When the overheads are much greater than the execution costs of the tasks, the allocation of the critical path to a processor is very inefficient. Therefore, it becomes necessary to cluster adjacent tasks so that the total number of communication events is decreased. The *layer* partition provides subtrees as clusters of tasks, producing a coarser *DAG*. However, depending on the value of λ, the number of communication events which are incurred can be very high, since a $(\gamma+2)$-ary tree is produced.

In the light of the above discussion, we propose a partition for the Binary Tree in which the number of communication events is not dependent on the overhead parameter. The intuitive idea behind the partition is to divide the Full Binary Tree B_n *vertically* and each subtree is allocated to one of the m processors. The m subtrees are executed in parallel so that no communication event is incurred. However, the Binary Tree B_n is not completely divided in parallel subtrees. At some level of B_n towards the source task, each path is allocated to a distinct processor. The proposed allocation is considered as a *hybrid* one, since two allocation patterns are identified. We shall refer this hybrid allocation as the *subtree* partition, which is fully described next.

To keep the explanation simple, we assume that $m = 2^k$, for some k. The $\frac{n+1}{2}$ sources of B_n are divided into m subsets such that $\frac{n+1}{2m}$ tasks are sources of m subtrees, each subtree being allocated to a distinct processor. Each one of the m subtrees has its respective sink at level $k = \log m$.

We now define an allocation for the tasks located between levels 0 to $\log m - 1$ in B_n which are not in the subtrees, tasks. Each task v_l at level $\log m - 1$ is the immediate successor of the sinks of two distinct subtrees, each of which is allocated to distinct processors p_i and p_j. Let p_i be allocated to the left subtree and p_j the right one. Then, v_l is also allocated to processor p_i. Each task at level i, $i = 0, \ldots, \log m - 2$, is allocated to the same processor as its left immediate predecessor.

In Figure 9.3, the binary tree B_{31} is mapped to eight processors according to the *subtree* partitioning. The number i inside each vertex represents the vertex v_i and the processor to which v_i is allocated is written beside it.

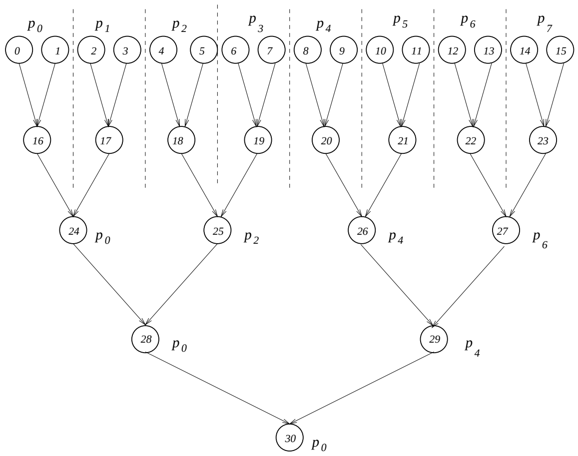


Figure 9.3: The *subtree* partition of B_{31} using $m = 8$ processors.

In the following Lemma, we are able to find an expression for the makespan of a schedule produced by the partitioning strategy that holds regardless of whether communication cost comprises latency or CPU penalty (overheads). We are able to do this because computation and communication is never overlapped due to the structure of the Binary Tree. Let $S_{subtree}$ be the schedule provided by the *subtree* partition.

Lemma 9.2.2 *The makespan of the subtree partitioning of a binary tree B_n with $l = \log(n+1)$ levels, considering a fully connected network with $m = 2^k$ processors, is*

$$\mathcal{M}(S_{Subtree}) = \frac{n+1}{m} - 1 + (\log m - 1)(Comm + 1)$$

where $Comm$ is the cost incurred when two processors communicate.

Proof:

The sinks of the m subtrees are at level k. The tasks at level k up to level $l - 1$ are divided between m processors into subtrees which do not communication among each other. Therefore, the number of tasks of each subtree (executed on a distinct processor) is

$$\frac{1}{2^k}\sum_{i=k}^{l-1} 2^i = \frac{(n+1)}{m} - 1$$

From level $k-1$ towards level 0, each task v_i is executed on the processor allocated to the left immediate predecessor v_l (see Figure 9.3), after communicating with its right immediate predecessor v_r, and therefore, v_i starts execution $Comm$ time units after the processor which was allocated to v_r initiates communication. Therefore, the sink task of B_n finishes execution at time

$$\frac{(n+1)}{m} - 1 + (\log m - 1)(Comm + 1) \tag{9.6}$$

□

Following the above Lemma, the number of processors necessary to achieve the minimum makespan produced by this partition is provided by evaluating $\frac{\delta\mathcal{M}(S_{Subtree})}{\delta m} = 0$, i.e.

$$m_{opt} = \lfloor \frac{n+1}{Comm+1} \ln 2 \rfloor$$

To characterise communication, the two parameters considered are λ and τ. The cost of the communication $Comm$ incurred between two communicating processors is due to a precedence relationship between the unique tasks allocated to each one of the processors, i.e. for each task $v \in B_n$ (which is not a source task), the right immediate predecessor $v_r \in V_B$ is allocated to a different processor. Therefore, at each level just one send and one receiving incurs between the execution of tasks at levels $\log m - 1$ to 0, added to the delay when the data item is being transfered. Therefore, $Comm = 2\lambda + \tau$.

9.2.3 Applying MSA to the Binary Tree

The following Lemma establishes that the *multi-stage approach* can produce a schedule for the Binary Tree which is equivalent to the *subtree* partition if the *clustering factor* γ given as input has value zero.

Lemma 9.2.3 *Let $B_n = (V_B, E_B)$ be the Binary Tree given as input to the first and second stages of the multi-stage approach. Let $\gamma = 0$ be the clustering factor given to the replication algorithm in the first stage. The final schedule produced by the remapping algorithm, given m physical processors is equivalent to the one produced by the subtree partitioning.*

Proof:

When executing the *replication* algorithm, given the Binary Tree B_n and $\gamma = 0$, each cluster produced contains a single task and each cluster is allocated to a distinct *virtual* processor. The set of clusters produced are in the set $V_{cluster}$. As each cluster correspond to a task, we refer to them as task rather than cluster or supertask.

The $superdag$ algorithm is applied to $V_{cluster}$. For any task $v \in V_{cluster}$, the list $\mathcal{I}(v)$ coincides with $pred(v)$. As the tasks are not replicated, there exists only one cluster containing each one of the tasks $u \in \mathcal{I}(v)$. Thus, the *Unique Copy* Rule establishes the superedge (edge) between v and its immediate predecessors. Therefore, the superdag $G^* = (V^*, E^*)$ produced by the $superdag$ algorithm is also a binary tree (G^* is isomorphic to B_n).

When executing the second stage given $m = 2^k$, λ and τ, each band to be evaluated by the *remapping* algorithm coincides with each level of G^*. Therefore, the number of tasks reallocated to one processor at level β of G^* is $\eta(\beta) = 2^{\beta-k}$, for $i = 0, \ldots, l-1$, since Brent's Principle is applied.

The *remapping* algorithm starts analysing the sink level correspondent to the band $\beta = 0$. Since this is the first supertask to be reallocated, the $free\_allocation$ procedure is performed, mapping the sink task to a processor, say p_0.

Let level $\beta = 1$ (with 2^1 tasks) be evaluated and let $k = \log m > 1$, so that 2 processors are used to allocate one task each. The *remapping* algorithm establishes that $lproc(\beta) = \{p_0\}$, since this processor was the only one which participated in the reallocation process during the previous band. The *common-immediate-successor* mechanism is applied and only one of the two immediate predecessors of the sink task is allocated to the processor p_0, since $\eta(\beta) = 1$. Then, $free\_allocation$ is executed, because $\mathcal{C}(\beta)$ is not empty and this procedure allocates the immediate predecessor of the sink supertask to any processor other that p_0.

Let β be the next band to be reallocated, and also, let $k > \beta + 2$. Therefore, $\eta(\beta) = \left\lceil \frac{2^\beta}{2^k} \right\rceil = 1$. Let the set of processors which participated in the reallocation process during iteration $\beta - 1$ be $lproc(\beta) = \{p_0, p_1, \ldots, p_i\}$, and obviously, $\eta(\beta - 1) = 1$. Suppose that, w.l.o.g, the task (supertask) v_i is reallocated to the processor p_i during iteration $\beta - 1$. Therefore, the *common-immediate-successors* mechanism establishes that one of the immediate predecessors of task v_i is reallocated to the same processor p_i. Let this immediate predecessor of v_i be the left side one. This mechanism is applied for each processor in $lproc(\beta)$. When $lproc(\beta) = \emptyset$, the procedure $free\_allocation$ is performed since $\mathcal{C}(\beta) \neq \emptyset$, assigning the non-allocated immediate predecessors of v_i to a processor $p_j \neq p_i$. This p_j is not *full* and it has not been allocated to any task yet, becoming full through the assignment of only one task. Neither the *common-immediate-predecessor* nor the *successor-same-processor* mechanisms are applied, since the processors allocated to the left immediate predecessor of v_i has not yet participated in the reallocation process.

Suppose the level $\beta = k + 1$ is the next band of tasks to be reallocated. Each task at level $\beta - 1$ was allocated to one of the m available processors. Suppose, w.l.o.g., that task v_i is allocated to processor p_i, $\;i = 0, \ldots, 2^{\beta-1}$. At level β, the number of tasks to be allocated per processors is $\eta(\beta) = 2$ and therefore, the *common-immediate-successors* mechanism reallocates both immediate predecessors of v_i to the same processor p_i as v_i. During the reallocation of tasks at each subsequent level $\beta = k + 2$ to l, the value of $\eta(\beta) = 2^{\beta-k}$ respectively, and $lproc(\beta) = \{p_0, p_1, \ldots, p_{m-1}\}$. Consequently, the *common-immediate-successors* mechanism reallocates to each processor $p_i \in lproc(\beta)$, both immediate predecessors of all of the tasks reallocated to p_i during the previous iteration. As a result each one of the m processors is assigned to the subtree whose sink is at level k.

$\square$

MSA was performed for various Fully Binary Trees and it was certified that for $\gamma = 0$, the *subtree* partition is produced and the *layer* partition is provided when $\gamma \neq 0$.

9.2.4 Experimental evaluations

Overheads are the predominant communication costs

When executing the *subtree* partition onto m processors, the number of communication events that occur when two processors communicate is equal to two, regardless of the number of tasks, the value of λ and the number of processors m. On the other hand, in the *layer* partition with $\gamma = \lambda$, the binary tree is "transformed" into an $(\lambda + 2)$-ary tree, corresponding to the fact that $(\lambda + 1)$ receiving events are carried out and therefore, the higher the value of λ, the smaller the number of processors that can usefully be applied to achieve the minimum makespan.

Figure 9.4 shows the comparison between the two partition strategies for the B_{2046} tree with 11 levels, considering $\lambda = 2$. The layer partition produces results very close to the *subtree* partition, because, the in degree of each non-source supertask in the resulting superdag is 4. On the other hand, in Figure 9.5, the results from both strategies considering the same input tree, but $\lambda = 30$ shows that for a large number of processors, the *layer* partition produces a schedule that is very far from the optimal one. Therefore, the largest is the startup overhead associated with the sending and receiving of messages, the worse is the schedule provided by the *layer* partition when comparing with the schedule provided by the *subtree* partitioning.

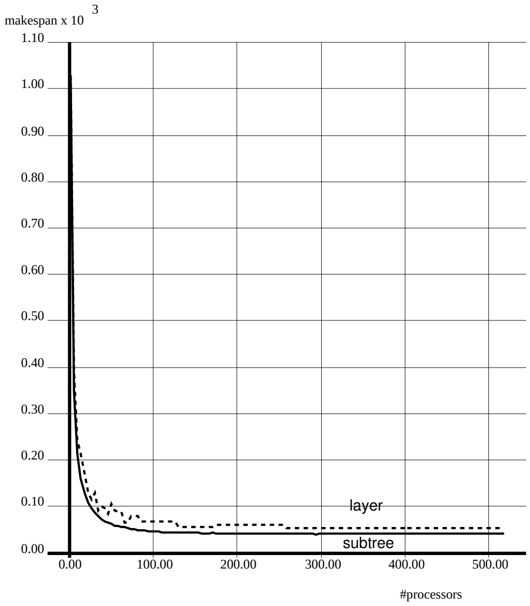


Figure 9.4: The makespan of the *subtree* ($\gamma = 0$) and *layer* ($\gamma = \lambda$) partitions for B_{2047} with $\lambda = 2$ and $\tau = 0$.

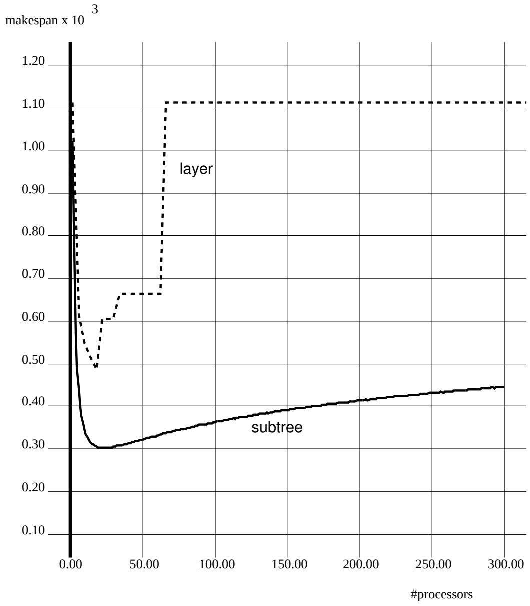


Figure 9.5: The makespan of the *subtree* ($\gamma = 0$) and *layer* partitions ($\gamma = \lambda$) for B_{2047} with $\lambda = 30$ and $\tau = 0$.

The *delay* model

Suppose that the number of available physical processors are at least $m_l = \frac{n+1}{\tau+2}$. In order to compare the *subtree* and the *layer* partitions considering the *delay* model, m_l is substituted in Equation 9.6, which expresses the makespan for the schedule produced by the *subtree* partition $\mathcal{M}(S_{Subtree})$. As a result, the following expression:

$$\log(\frac{n+1}{\tau+2})(\tau+1) \tag{9.7}$$

is the makespan of $S_{Subtree}$ when $m = \frac{n+1}{\tau+2}$. In order to derive the number of tasks n of B_n and of the value τ for which the *subtree* partition gives a smaller makespan than $\mathcal{M}(S^{delay}_{layer})$, the inequality $\mathcal{M}(S_{subtree}) < \mathcal{M}(S_{layer})$ is evaluated, taking into account that m_l processors. Using Equations 9.7 and 9.1, we have that

$$\log(n+1) < \frac{\lceil \log(\tau+2) \rceil}{1 - \frac{1}{\lceil \log(\tau+2) \rceil - \frac{\tau}{\tau+1}}}$$

which shows that the higher is τ, the smaller must be the size of the input B_n so that the *subtree* partition produces a better schedule. In Figure 9.6, the input binary tree is the B_{10} and the only communication parameter τ with value equal to 2. In such case, even when $m_l = 256$, $\mathcal{M}(S_{subtree}) \approx \mathcal{M}(S^{delay}_{layer})$, however, the *layer* partition is the best approach. The experiments assured that the *layer*

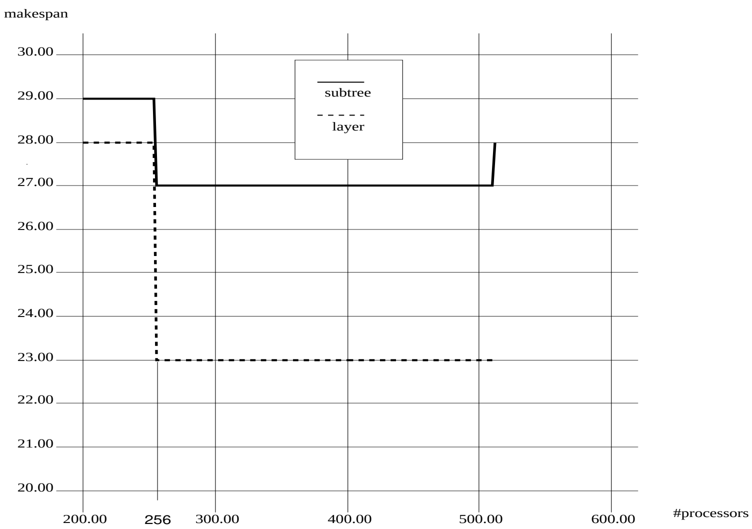


Figure 9.6: *subtree* ($\gamma = 0$) and *layer* ($\gamma = \tau$) partitions for B_{1024}, $\tau = 2$ and $\lambda \simeq 0$; $m_l = 256$.

partition provides schedules with a smaller makespan that the *subtree* partition, mainly when $m = \frac{n+1}{\tau+2}$. In Figure 9.7 the input is the Binary Tree B_{2047} and $\tau = 6$. The best number of processors for the *layer* partition in this experiment is $m_l = 256$, and we clearly see the benefits of partitioning the tree into layer. For a smaller number of processors, the *subtree* partition may produce a good schedule but not with makespan smaller than $\mathcal{M}(S^{delay}_{layer})$. In Figure 9.8 the input tree is also B_{2047} but the τ value is 30 and so, $m_l = 64$. For such higher delay, the benefit of using the *layer* partition is clearly verified.

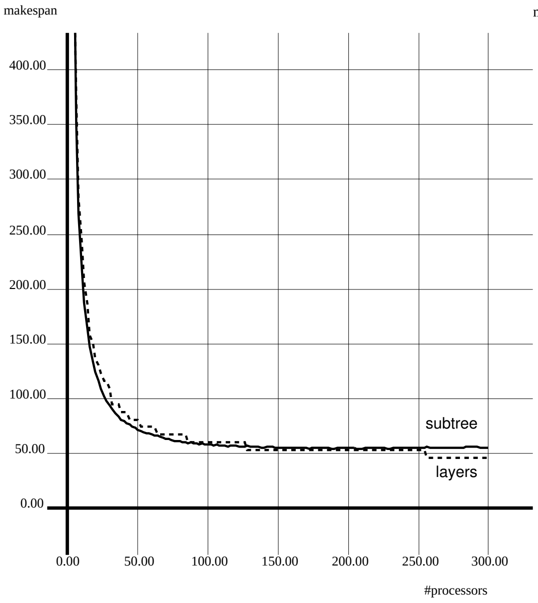


Figure 9.7: The makespan of the *subtree* ($\gamma = 0$) and *layer* ($\gamma = \tau$) partitions for B_{2047}, $\tau = 6$ and $\lambda \simeq 0$; $m_l = 128$.

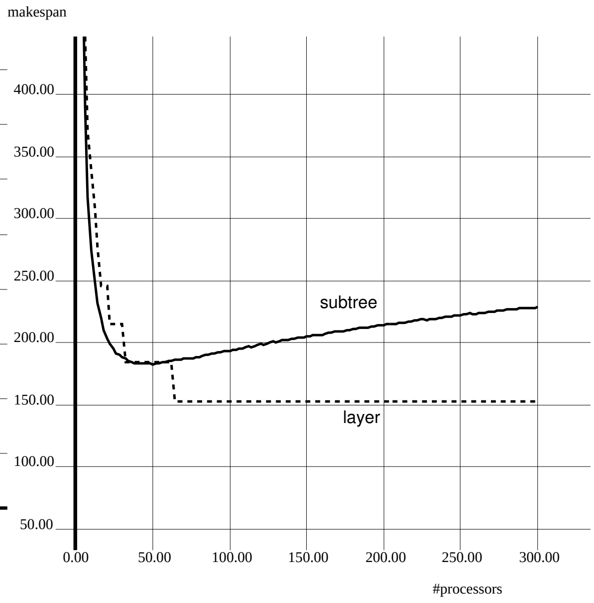


Figure 9.8: The makespan of the *subtree* ($\gamma = 0$) and *layer* ($\gamma = \tau$) partitions for B_{2047}, $\tau = 30$ and $\lambda \simeq 0$; $m_l = 64$.

The CLAUD model

Both communication parameters are now considered significant. Two situations are given attention in the experiments:

1) $\lambda > \tau$**:** since the overhead is the dominant communication parameter cost, it is not good to cluster tasks into a supertasks, as in the *layer* partition, because the in degrees can be large. Regardless the given number of processors, the best approach is the application of the *subtree* partition. Figure 9.9 shows the comparison of the 3 partitions using $\gamma = 0, 2, 6$, $\tau = 1$ and $\lambda = 6$ for the Binary Tree B_{511} (9 levels and 256 sources). As we see, in all the cases, the *subtree* partition gives a smaller makespan.

2) $\lambda < \tau$**:** consider the case in which $\lambda \leq \tau/2$ (the factor of two is due to the fact that a full subtree can be clustered into a supertask). The delay is the predominant factor on the communication costs and although for $\gamma = \tau$ the respective superdag may have a high in degree, the values for λ are not high enough to hinder the benefits of partitioning the Binary Tree into layers. The best results were achieved when the number of processors is $m \geq \frac{n+1}{\tau+2}$ and $\gamma = \tau$ for the *layer* partition. Figures 9.10 and 9.11 shows that for B_{511}, the *subtree* partition provides supertasks with grains that are smaller than the delay, even adding the overheads to the execution cost of the tasks.

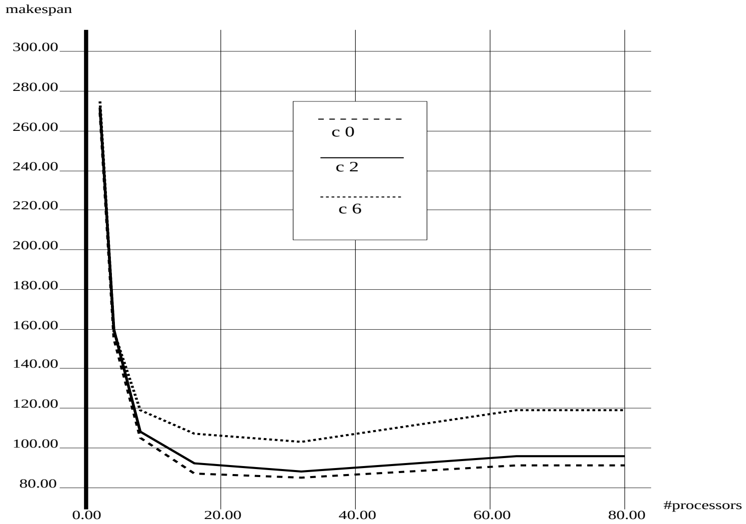


Figure 9.9: *subtree* ($\gamma = 0$ corresponds to $c0$) and *layer* partitions ($\gamma = 2, 6$, correspond to $c2$ and $c6$, respectively). The input are: $B_5 11$, $\tau = 1$ and $\lambda = 6$.

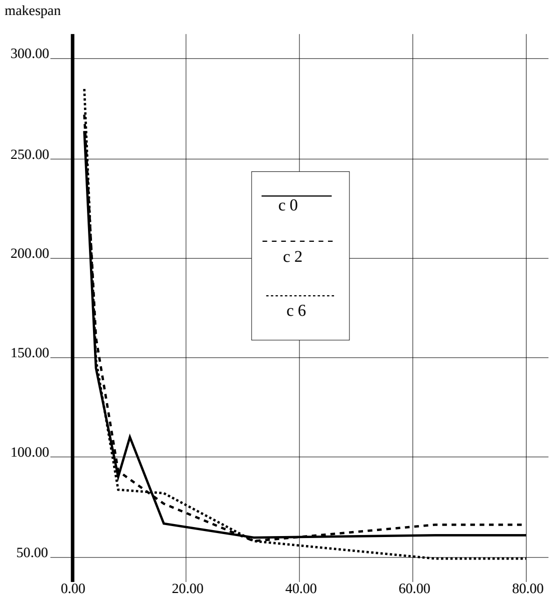


Figure 9.10: *subtree* ($\gamma = 0$, correspondent to $c0$) and *layer* ($\gamma = 2, 6$, correspondent to $c2$ and $c6$, respectively) partitions for B_{511}, $\tau = 6$ and $\lambda = 1$.

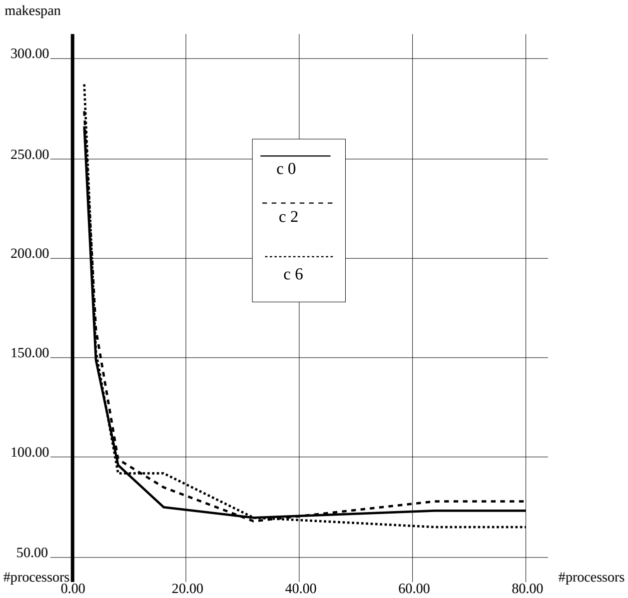


Figure 9.11: *subtree* ($\gamma = 0$, correspondent to $c0$) and *layer* ($\gamma = 2, 6$, correspondent to $c2$ and $c6$, respectively) partitions for B_{511}, $\tau = 6$ and $\lambda = 2$.

MSA and other strategies - an experimental evaluation

The best results produced by the *layer* and *subtree* partitions for the Binary Tree B_{511}, are compared with ETF. We have seen that in the case in which $\lambda > \tau$ it is better to set the parameter γ to zero in the first stage of MSA so that the *subtree* partition is produced. In Figure 9.12 the results of MSA and ETF are compared, given that $\lambda = 6$ and $\tau = 1$. One of the problems with ETF is that due to local decisions, when the number of available processors is smaller than the number of source tasks , source tasks which do no belong to the same subtree may be allocated to the same processor. ETF does not attempt to minimise the number of communication events and therefore, communication events are incurred at each level of B_{511}, regardless the number of processors considered. Concluding from the observations, the ETF schedule has worse properties when compared with MSA.

The second case examines the situation in which $\lambda < \tau$. MSA is executed considering $\gamma = \tau$. In Figure 9.13, ETF and MSA results are shown for $\lambda = 1$ and $\tau = 6$. The ETF algorithm results are closer to the *subtree* partition, but only because the startup overhead is much smaller than the delay τ and also, due to the structure of the Binary Tree, it is easier to precisely predict the schedule of the *communication event* tasks.

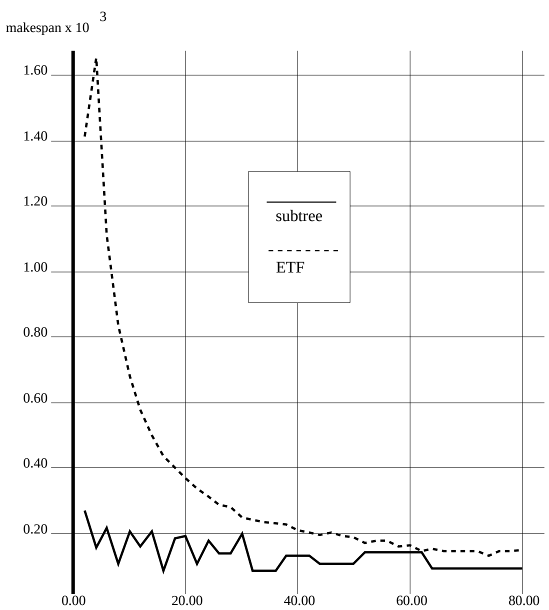


Figure 9.12: The *subtree* partition ($\gamma = 0$ using MSA) and ETF results for B_{511}, $\tau = 1$ and $\lambda = 6$.

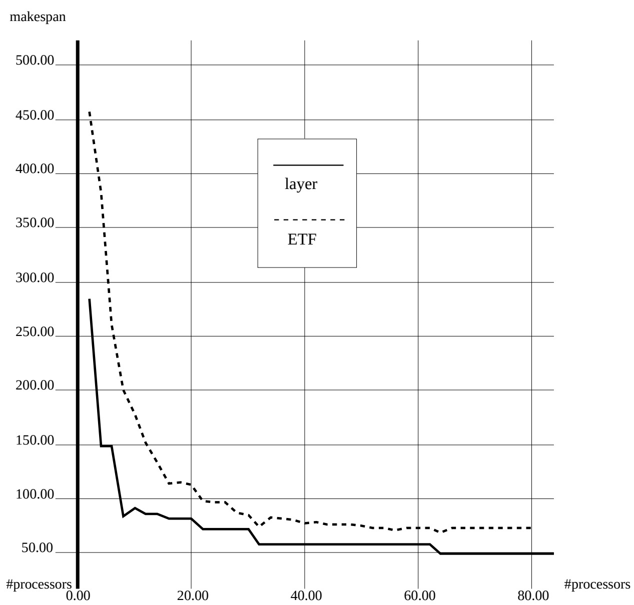


Figure 9.13: The *layer* partition ($\gamma = \tau$ using MSA) and ETF results for B_{511}, $\tau = 6$ and $\lambda = 1$.

We execute the schedules provided by MSA and ETF for B_{511} on the simulated machine considering a fully connected network. Both predictions are very accurate mainly due to the binary tree structure where each task has only one immediate successor. However, as show in Figure 9.14 where $\lambda = 6$ and $\tau = 1$, MSA gives better results than ETF for any number of processors and also, predicts very precisely.

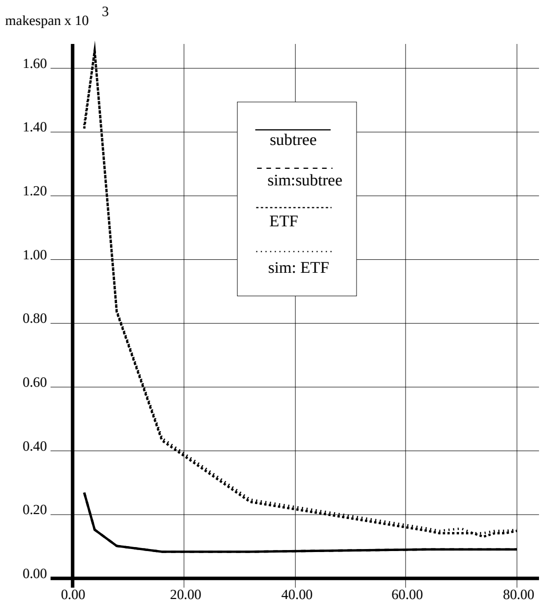


Figure 9.14: The *subtree* partition ($\gamma = 0$ using MSA) and ETF results for B_{511}, $\tau = 1$ and $\lambda = 6$.

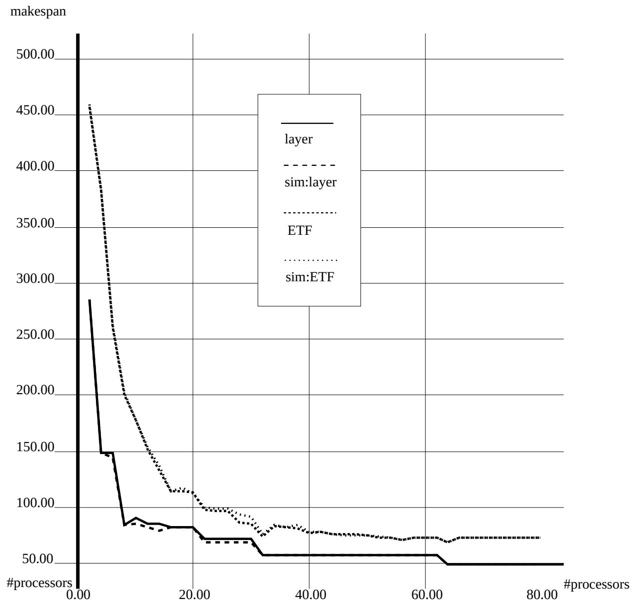


Figure 9.15: The *layer* partition ($\gamma = \tau$ using MSA) and ETF results for B_{511}, $\tau = 6$ and $\lambda = 1$.

The problem with ETF starts during first step of the strategy: it allocated each source task to a distinct processors, without taking into account the adjacency of the task yet to be scheduled. As the startup overhead is high, the resulting schedule has high value.

In the case that the delay is greater than the startup overhead, the predictions of both approaches are very close to each other. but still MSA produces better results, as seen in Figure 9.15, where $\tau = 6$ and $\lambda = 1$. Both prediction and simulations of MSA are smaller than those correspondent to ETF results.

In order to compare the *multi-stage approach* with MH, experiments are carried out in which the significant communication parameters are the *sending overhead* and *delay*, since MH does not consider *receiving* overheads. Considering the Full Binary Tree B_n the *sending event* tasks are executed in parallel, either partitioning B_n by *layers* or applying the *subtree* partition technique. Therefore, the best partition technique is the one that minimises the number of *sending event* tasks along the critical path of the correspondent superdag. In this case, the in degree of the $(\gamma + 2)$-ary tree produced by the *layer* partition does not hinder the performance of the resulting schedule.

Firstly, experiments are performed taking into account the delay as the only significant parameters. The time to completion of the simulation of MSA schedules are the same as predicted and therefore, they are omitted from the respective Figures. The best value of γ are also derived experimentally, which is related with the communication costs. The best γ is the value which produces supertasks that are full sub-binary trees. Figures 9.16 and 9.17 shows the comparison when $\tau = 6$ and $\tau = 10$, respectively. We see that for higher τ values, the difference between the makespan of MH schedules and the makespan of MSA schedules is greater. Observe that when the number of processors is small, MH may produce better results those produced by MSA, if the delay is not very high. In all of the cases considering the *delay* model, MSA produce better results then MH.

The next set of experiments are those in which λ_s and τ are significant. Again, since MSA predicts very accurately, we omit the time to completion of the simulation. This fact is not the same for MH. Figures 9.18 and 9.19 shows that as the values of λ_s increases, the difference between the time to completion of the simulation and what was predicted also increases, in the case of MH.

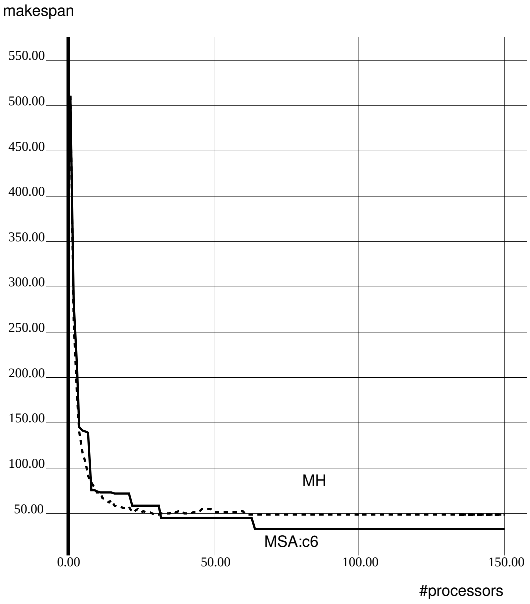


Figure 9.16: Results of MSA ($\gamma = 6$) and MH, considering B_{511}, $\tau = 6$, $\lambda_s = \lambda_r = 0$.

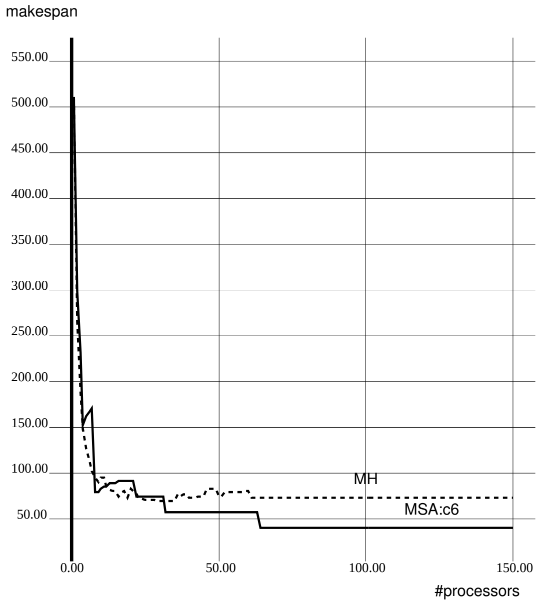


Figure 9.17: Results of MSA ($\gamma = 6$) and MH, considering B_{511}, $\tau = 10$, $\lambda_s = \lambda_r = 0$.

9.2.5 Concluding remarks

Jung *et al.* [70] showed that the Full Binary Tree can be asymptotically optimally scheduled without recomputation, considering that the target model has as single communication parameter the delay τ. The *layer* partition provides a mechanism of transforming the input *DAG* onto a coarser one, by the specification of clusters of tasks equivalent to subtrees. The degree of parallelism is exploited when executing the subtrees in parallel at each layer. Gao *et al.* [44]

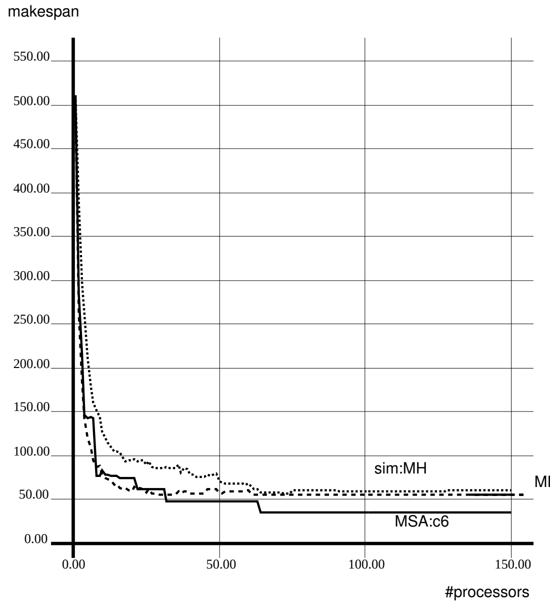


Figure 9.18: Results of MSA for $\gamma = 6$ and MH, considering B_{511}, $\tau = 6$, $\lambda_s = 1$ and $\lambda_r = 0$.

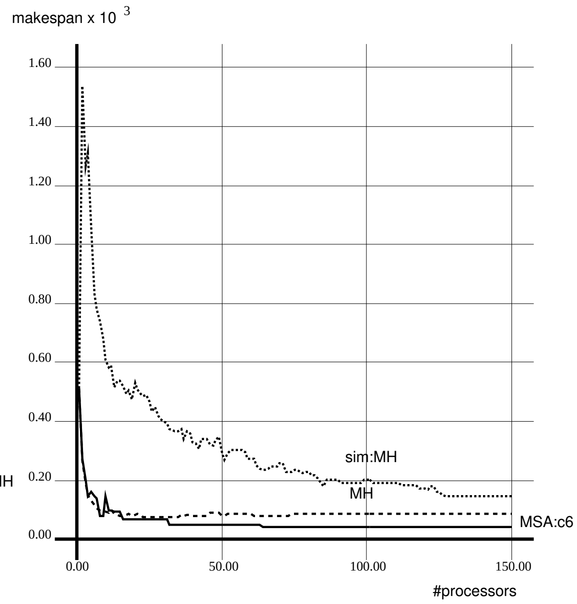


Figure 9.19: Results of MSA for $\gamma = 6$ and MH, considering B_{511}, $\tau = 1$, $\lambda_s = 10$ and $\lambda_r = 0$.

proposed an algorithm that produces an optimal schedules for fine grained B_n in the sense that $\tau > 1$ and assuming the *delay* model and an unbounded number of processors. They do not have a closed expression for the best makespan, but they prove that their schedules are optimal.

However, when the *blocking* parameter λ (λ_s and λ_r) the partitioning of the tree into layers produces supertasks with in degree greater than or equal to 2. In architectures with high overheads, the parallel execution of such superdag leads to a high communication cost due to the sending and receiving events.

The *subtree* partitioning is proposed to overcome this problem. This partitioning technique is provided by MSA given $\gamma = 0$ as the *clustering factor* to the *replication* algorithm. Therefore, it is not during the first stage that the input Binary Tree is transformed onto a coarser *DAG*. The tasks are clustered on a “vertical” fashion rather then “horizontal”, as the one in the *layer* partition by the *remapping* algorithm during the second stage, where the parameter m guides the clustering.

We note that, although MSA produces better results than ETF and MH for the Fully Binary Tree mainly when τ has a high value, ETF and MH do not provide schedules that are completely unreasonable considering this kind of *DAG*. For the *delay* model, both ETF and MH can provide better results than MSA when there are small number of available physical processors and low τ. The same does not hold for very large delays. The existence of overheads shows how MH and ETF are not realistic. Mainly MH, a scheduling heuristic that specifies the startup overhead when sending a message and the delay on communications as communication parameters. In this case, MH can produce schedules with makespan smaller than those provided by MSA, when the sending overhead is not very high and the number of processors is small. For higher communication costs (mainly sending overheads), MH predictions are higher than MSA. However, the time to completion of the simulations shows how unrealistic are the assumptions assumed by MH. The decisions performed by MH have as main objective the minimisation of the makespan by scheduling and allocating tasks in such a way that the computation of tasks is overlapped by communication. By adding the startup overhead to the delay, MH specifies that the startup cost can also be overlapped with task computation. This is completely unrealistic and consequently, the time to completion of the simulation is much higher than the respective predictions.

9.3 Full Inverted Binary Tree

The Inverted Binary Tree $I_n = (V_I, E_I)$ is a *DAG*, where the tasks correspond to the set of vertices V_I and the precedence relationship between tasks is represented by the set edges in E_I . The edges are directed from one single source task or *root* towards the leaves, which are the sink tasks. Each task, apart from the leaves, has exactly two immediate successors and also, each task apart from the root, has exactly one immediate predecessor. The main characteristics and notations of the Inverted Binary Tree are given as follows: the height of I_n is $l = \log(n+1)$; the levels of the inverted tree are defined from the root, which is at level 0, to the leaves, which are at level $l-1$; the number of tasks at level i is 2^i, $0 \le i < l$; I_n has $\frac{n+1}{2}$ leaves; there is one single path from the root to each other vertex $v_i \in V_I$.

An evaluation on the schedule of the Inverted Binary Tree considering a network with m processors is carried out, taking into account the startup overhead λ and the delay on the communication between adjacent processors τ. In this evaluation, the tasks has unit execution cost and through each edge, one data item is transmitted, such that the paths from the root to the leaves have the same length.

Lemma 9.3.1 *In order to achieve the optimal schedule of the Inverted Binary tree $I_n = (V_I, E_I)$ onto a network of $m \ge 2^{l-1}$ processors under any topology, each path from the root to each one of the leaves is replicated to a distinct processor* *together with the respective leaf task.*

Proof:
Each path from the root to any of the leaves v_i is $\log(n+1)$ long. Suppose the root v_r and two leaves v_k and v_l are such that the respective paths $c_k = \langle v_r, \ldots, v_i, \ldots, v_k \rangle$ and $c_l = \langle v_r, \ldots, v_i, v_j, \ldots, v_l \rangle$. Suppose that all tasks in c_k are allocated to a processor p and the tasks in the sub-path $\langle v_j, \ldots, v_l \rangle \subset c_l$ where $\langle v_j, \ldots, v_l \rangle \not\subset c_k$ are allocated to a processor $p' \neq p$, so that no task is replicated. Therefore, after p executes the task v_i, it sends the result to p' in order to execute task v_j, incurring a communication cost of $2\lambda + \tau$. Therefore, the resulting length of the schedule of the path c_k is $\log(n+1) + 2\lambda + \tau$. On the other hand if $\langle v_r, \ldots, v_i \rangle$ is replicated on processor p', the schedule length of c_k is $\log(n+1)$. The same follows for all the other paths.

Under such replication strategy, no communication is carried out among processors, and so, topological details of the target machine can be ignored.

$\square$

The minimum makespan $\log(n+1)$ is achieved when the number of available processors is $m = 2^{l-1}$ (number of leaves). It is obvious that if less processors are physically available, the resulting makespan is higher. Suppose $m = 2^k$ for some $k < (l-1)$. Therefore, the tasks at level k (see Figure 9.20) are roots of the subtrees whose tasks are allocated to each one of the m processors where each respective subtree is executed sequentially. The makespan of the schedule of I_n onto m processors is:

$$\log m + \frac{2^l - 1 - (2^k - 1)}{m} = \log(m) + \frac{n+1}{m} - 1 \tag{9.8}$$

9.3.1 Applying the *multi-stage approach*

In order to find the optimal solution, the Inverted Binary Tree I_n is given as input to the first stage, in which the clustering of tasks through replication is carried out. However, rather than assigning the *clustering factor* to the overhead of sending or receiving a message, or even to the delay value, γ is assigned to $\log(n+1) - 1$, so that the optimal makespan is achieved (each task has only one immediate predecessor, so each critical path forms a new supertask). After executing the first stage, a superdag without superedges is produced. The superdag is given as input to the second stage which is characterised by the application of Brent's Principle. There is no use of the *common-immediate-successors*

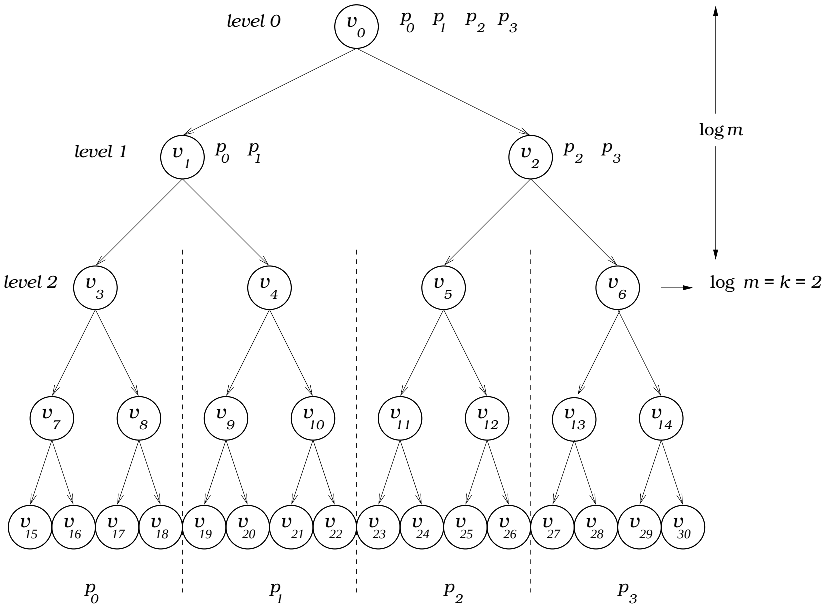


Figure 9.20: Inverted Binary Tree I_{31} scheduled on $m = 4$ processors with task replication

and *common-immediate-predecessors* mechanisms, because the input superdag has no superedges. The number of supertasks reallocated to each processor is then $\frac{l2^{l-1}}{m}$, as defined by Brent's Principle. Note that in the schedule produced by the first stage, there exists one single band (a computational interval), since no communication is carried out among supertasks. There exist 2^{l-1} parallel supertasks, each with l tasks, ordered in accordance with their precedence relationship. However, many copies of the same tasks may be reallocated to the same processor and obviously, these copies are reduced to a single one. Therefore, the makespan of the schedule produced by MSA of I_n for m processors is the one derived in Equation 9.8.

9.3.2 Concluding remarks

Jung *et al.* [70] derived a general lower bound considering that replication of tasks can be limited for the *delay* model. Considering a schedule S in which has at most c computations per task, they proof in a theorem in [70] that the makespan of S is at least

$$\tau \frac{\log(n+1)}{2(\log(n+1) + \log c)}$$

The limitation on the amount of replication considering the CLAUD model corresponds to the specification of *communication event* tasks. When a smaller γ is used, $b = \frac{\log(n+1)}{\gamma+1}$ *computational* intervals are defined. In each interval

$\beta = 1, \ldots, b$, a task at level j of I_n is replicated

$$\frac{2^{\beta(\gamma+1)-1}}{2^j}$$

in its respective interval. Due to the structure of I_n, no more than $2^{\gamma+1} - 1$ *sending event* tasks and no more than one *receiving event* task are incurred in each *communication* interval. Added to that, the delay on communication associated with each *communication* interval is no more than τ.

Also, the parameter m can be used in order to limit the amount of replication, if this restriction apply. We know that the reallocation to a number of processors smaller than $\mathcal{W}(G^*)$ will force more tasks to be executed to each physical processor, but less replication and *communication event* tasks are incurred (maybe no *communication event* task, such as the discussion that led to Equation 9.8).

We conclude that when replication is forbidden, or at least restricted, in systems where the overheads are significant, MSA can be applied by tuning the parameters γ and m in accordance with the requirements. However, depending on the values of λ_s and λ_r, it may be better to execute the application on a single processor when trying to decrease the amount of replication.

9.4 The *diamond DAG*

We wish to analyse the benefits provided by the *replication* algorithm when scheduling the diamond *DAG*. The algorithm incorporates the *higher-number-of-immediate-successors* mechanism, which may provide more regularity on the definition of the clusters. This mechanism is the first attempt to decrease the number of *communication events*, mainly the *receiving event* tasks, implemented in the overall strategy.

In the very first version of the *replication* algorithm there was no concern to cluster tasks which are highly connected (high exchange of messages). As seen in Figure 9.21, when the $superdag$ algorithm tries to minimise the out degree of the resulting supertasks, more clusters may be selected as immediate predecessors. As a result, the superdag produced (Figure 9.21) has many supertasks and some of them have a relatively high in degree. The only solution to decrease the number of *receiving event* tasks is to use a higher *clustering factor*. The result for $\gamma = 11$ is shown in Figure 9.22.

Not only the *clustering factor* is related to the communication costs, but also, each cluster should contain the maximum number of immediate predecessors of

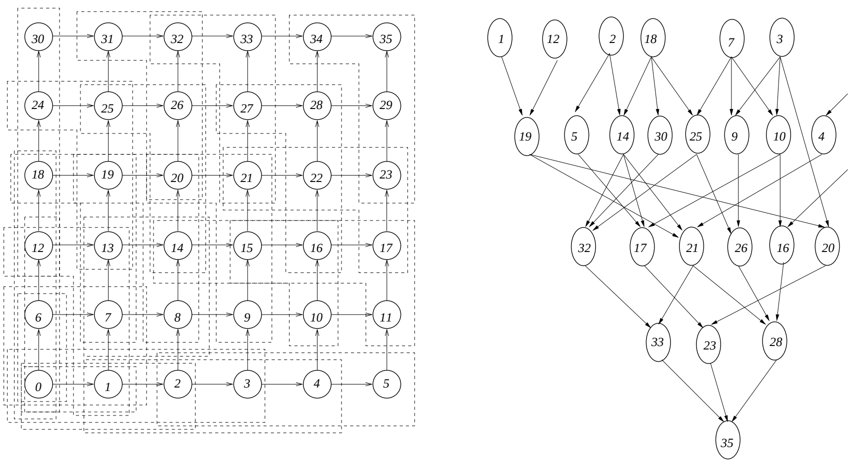


Figure 9.21: Clusters produced when the *higher-number-of-immediate-successors* mechanism is not used for $D_{6\times6}$ and $\gamma = 3$. On the right side, the correspondent superdag.

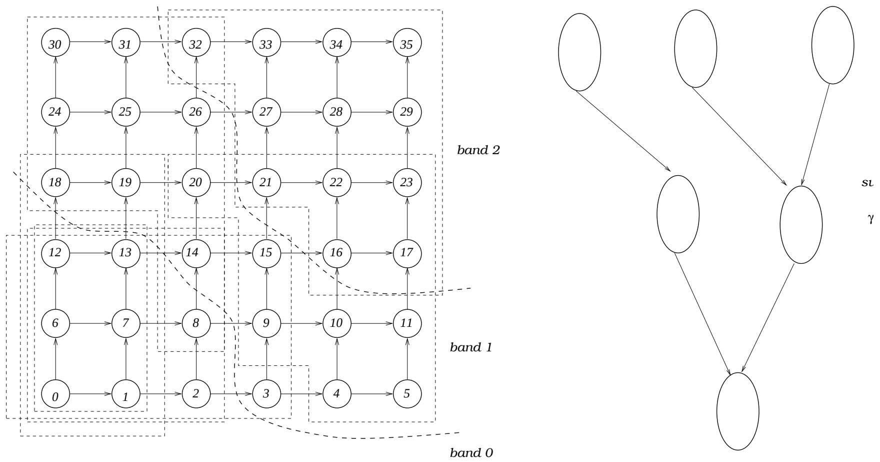


Figure 9.22: Clusters produced when the *higher-number-of-immediate-successors* mechanism is not used for $D_{6\times6}$ and $\gamma = 11$. On the right side, the correspondent superdag.

tasks inside clusters scheduled at subsequent bands. In the case of the diamond *DAG*, a *box*-like partition would produce a schedule with a smaller makespan, considering that one of our goals is to minimise the number of *communication events* [104].

The *replication* algorithm, as specified in Section 5.3, incorporates the *higher-number-of-immediate-successors* mechanism and also the e_l-values associated with the tasks. The same $D_{6\times6}$ is given as input to the *replication* algorithm and, as shown in Figure 9.23, each supertask is exactly a *box*. However, it is important to clarify that it is not always the case in which clusters are exactly *boxes*, since the first priority used to construct the cluster is the e_l-value, and the second priority is the *highest-number-of-immediate-successors*. In Figure 9.24, the 8×8 diamond *DAG* and $\gamma = 15$ are given as input to the *replication* algorithm. The tasks with the same e_l-value are separated by the thick lines. When construct-

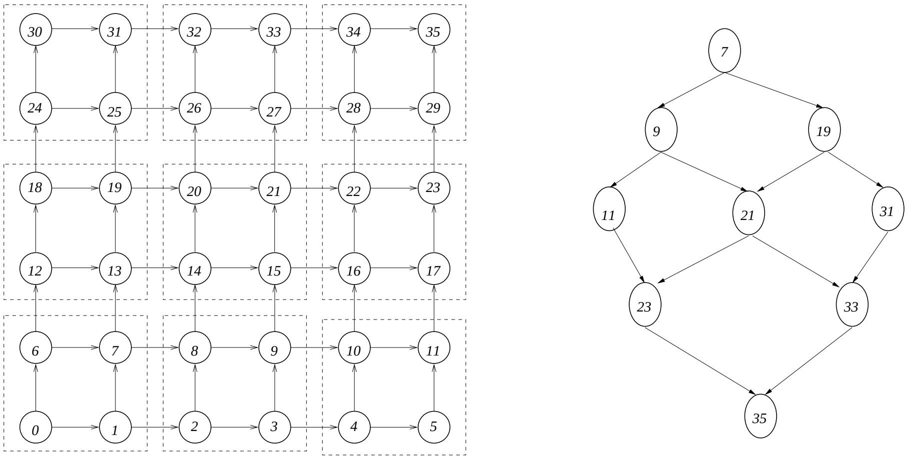


Figure 9.23: The supertasks produced by the *replication* algorithm for $D_{6\times6}$ and $\gamma = 3$. On the right side, the correspondent superdag.

ing the cluster in which the task 60 is the owner, although the task 49 has a higher number of immediate successors already in 60-cluster than the task 28, $e_l(28) > e_l(49)$. Therefore, task 28 is included in the 60-cluster, otherwise the precedence relationship between tasks allocated to different clusters would not be preserved. After the *replication* algorithm, the $superdag$ algorithm is per-

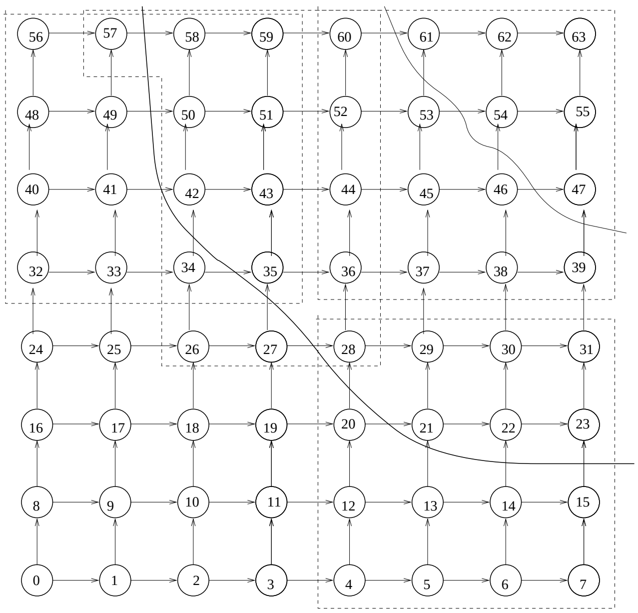


Figure 9.24: The *replication* algorithm executed for the $D_{8\times8}$ with $\gamma = 15$

formed. When the 63-cluster is visited (the sink supertask, which has exactly a *box* shape), during the first iteration of the first phase, the 60-cluster is selected as an immediate predecessor supertask, since it sends a larger amount of data than the 59-cluster.

The *boxes* partition [104, 105] is evaluated in the next section, considering the CLAUD model. The objective of these evaluation is to compare this partition and the schedule produced by the *multi-stage approach*. For the analysis, the number of *virtual processors* is given, which is denoted by k, which is equivalent to the number of resulting clusters (the *boxes*). We wish to experimentally compare the results of MSA with *boxes* partition and to identify similarities between the two approaches.

9.4.1 The *boxes* Partition

Let k be the number of clusters to which the $n \times n$ diamond *DAG* $G = (V, E)$ is partitioned. Each one of the k clusters consists of $\frac{n^2}{k}$ tasks. The *boxes* partition of a $n \times n$ diamond DAG in Figure 9.25 shows how tasks are clustered into *boxes* and consequently, a superdag can be defined considering the CLAUD model. Each supertask has a right and an upper immediate successors, apart from the last row (the top row in Figure 9.25) in which each supertask has just the right immediate successor. Each supertask has a left and a lower immediate predecessor, apart from the first row, in which supertasks have just the left immediate predecessor. The source supertask can be seen in the left bottom corner of the *super-diamond DAG* and the sink, in the top left corner. Each superedge has weight equal to $\frac{n}{\sqrt{k}}$, correspondent to the bundling of $\frac{n}{\sqrt{k}}$ unit length messages.

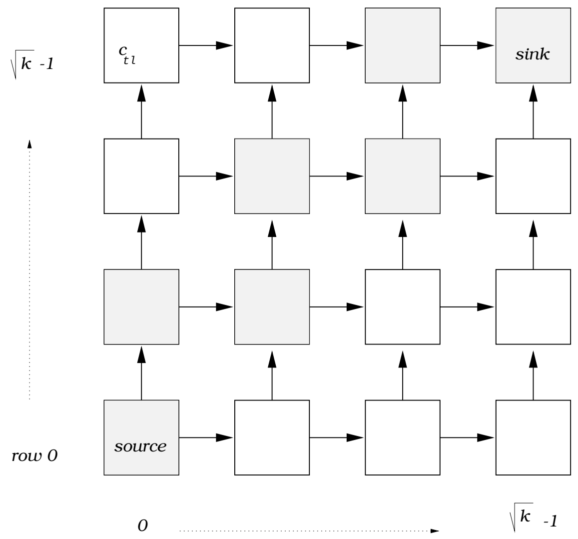


Figure 9.25: Boxes partition on the diamond *DAG*.

We initially specify the schedule on k virtual processors and $S^{virtual}_{boxes}$ is the set of tuples which defines the *virtual boxes* partitioning considering CLAUD:

***schedule order*:** each supertask starts being executed on a processor after the respective (if required) receiving event tasks are executed on the same processor. After the supertask finishes, the respective sending event tasks can be allocated to the same processor;

***allocation*:** each supertasks is executed on a distinct *virtual* processor.

The upper bound of $\mathcal{M}(S_{boxes}^{virtual})$ is derived taking into account λ_s, λ_r and τ. We say "upper bound", because depending on the values of the communication parameters, communication event tasks may have their execution overlapped with message transmission. The *critical path* of the $\sqrt{k} \times \sqrt{k}$ super-diamond *DAG* is shown in Figure 9.25. Through this path, the sum of computation times of its supertasks is maximal. Clearly, we can identify many *critical paths* on the super-diamond.

The upper bound on the computation time is $U_{comp} = (2\sqrt{k} - 1)\frac{n^2}{k}$. In a similar way, we can see that the cost associated with the superedges which connect the supertasks along the *critical path* corresponds to the total *delay cost* which is incurred on the overall computation. Each superedge sends a total of $\frac{n}{\sqrt{k}}$ data items and thus, the upper bound of the messages transmission time is $U_{comm} = (\sqrt{k} - 1)\frac{n}{\sqrt{k}}\tau$.

Finally, we need to consider the overheads λ_s and λ_r. Each supertask receives messages from no more than two immediate predecessors and sends to no more than two immediate successors, as seen in Figure 9.25. Therefore, the upper bound on the overhead time is $U_{over} = 2(2\sqrt{k} - 1)(\lambda_s + \lambda_r)$. Since in *critical path* there exists a supertask that does not receive messages, another one that does not send messages, one that receives only a single message and one supertask that sends only one message, we subtract $3(\lambda_s + \lambda_r)$ from U_{over}. As a result, the upper bound on the makespan is given by $U_{comp} + U_{comm} + U_{over} - 3(\lambda_s + \lambda_r)$, that is:

$$\mathcal{M}(S_{boxes}^{virtual}) = (2\sqrt{k} - 1)(\frac{n^2}{k} + \frac{n}{\sqrt{k}}\tau + (\lambda_s + \lambda_r)) - 3(\lambda_s + \lambda_r)$$

Examining this *virtual* allocation, the number of communication events may be decreased by applying the same principles and concepts used in MSA. On the mapping of the *boxes* on m *physical* processors, the set of processors $\mathcal{P} = \{p_0, p_1, \ldots, p_{m-1}\}$ is considered. We now specify the *physical boxes* partitioning of the super-diamond *DAG* and denote its schedule by S_{boxes}^{phys} considering the CLAUD model:

schedule order: each supertask starts being executed on a processor after the respective (if required) receiving event tasks are executed on the same processor. After the supertask finishes, the respective sending event tasks can be allocated to the same processor (also, if required);

allocation scheme: the allocation of the $\sqrt{k} \times \sqrt{k}$ *super-diamond DAG* is the following. The right immediate successor of a supertask c is allocated to the same processor $proc(c)$ and the upper immediate successor of c, is allocated to a distinct processor $p \neq proc(c)$.

Taking into account the overhead parameters λ_s and λ_r, and the latency τ are specified, an analysis of the schedule S^{phys}_{boxes} of the *super-diamond DAG* is discussed.

Let the source supertask c be allocated to processor p_0 at time step 0. After $\frac{n^2}{k}$ time units the sink task finishes to be executed and messages are sent to its immediate successors. Let c' and c'' be the upper and right immediate successors of c, respectively. As established by $allocation$, the supertask c'' is also allocated to p_0 and consequently, no cost is associated with the communication of the message (c, c''). The upper immediate successor c' is allocated to a distinct processor, say, p_1. Therefore, a *sending event* task $\sigma_{(c,cprime,p_0)}$ is scheduled at time $\frac{n^2}{k}$ on processor p_0, and then the respective message is transmitted at time step $\frac{n^2}{k} + \lambda_s$, taking $\frac{n}{\sqrt{k}}\tau$ time units to be transfered. A respective *receiving event* task $\rho_{(c,c',p_1)}$ is scheduled at time $\frac{n^2}{k} + \lambda_s + \frac{n}{\sqrt{k}}\tau$ on processor p_1 taking λ_r time units to be executed. Therefore,

$$s(c', p_1) = \frac{n^2}{k} + \lambda_s + \frac{n}{\sqrt{k}}\tau + \lambda_r$$

Let c_{tl} be the top leftmost supertask of the *super-diamond DAG* (as indicated in Figure 9.25). Following the schedule strategy above described, the start time of c_{tl} executed by the processor, say, p_j, is

$$s(c_{tl}, p_j) = (\sqrt{k} - 1)\frac{n^2}{k} + (\sqrt{k} - 1)(\lambda_s + \lambda_r) + (\sqrt{k} - 1)\frac{n}{\sqrt{k}}\tau \qquad (9.9)$$

after the *receiving event* task correspondent to the input message sent by the lower immediate predecessor allocated to processor p_j is executed.

In S^{phys}_{boxes}, each row i of the *super-diamond DAG*, $0 \leq i < \sqrt{k}$, is executed by a processor different from the one that executes row $i - 1$. Each supertask is executed after a *sending event* task (if required) and after the execution of each supertask, a *receiving event* is incurred. Each row $i = 2, \ldots, \sqrt{k} - 2$ takes

$\sqrt{k}(\lambda_s+\lambda_r+\frac{n^2}{k})$ to be executed, while row 0 is executed in not less than $\sqrt{k}(\lambda_s+\frac{n^2}{k})$. The execution of the last row is no more than

$$\sqrt{k}(\lambda_r+\lambda_s+\frac{n^2}{k}) \tag{9.10}$$

time units. Note that, although the last row does not send messages to other processors, it may stay idle for λ_s time steps each time that data is sent from the lower immediate predecessor of each supertask in the last row. The total makespan is provided by adding Equations 9.9 and 9.10 and subtracting $\lambda_s+\lambda_r$ time units, as the processor executing the first row does not spend time receiving messages, and the processor executing the last row has no *sending event* task allocated to it. Concluding the makespan of S^{phys}_{boxes} is

$$\mathcal{M}(S^{phys}_{boxes})=(\sqrt{k}-1)(\frac{n^2}{k}+\lambda_s+\lambda_r+\frac{n}{\sqrt{k}}\tau)+\sqrt{k}(\lambda_r+\lambda_s+\frac{n^2}{k})-(\lambda_r+\lambda_s)$$

Taking into account the allocation scheme, the maximum number of useful physical processors can be derived. Each row of supertasks is executed by a unique processor. When this processor finishes executing all supertasks of a row, it can be allocated to a yet to be scheduled row. Equation 9.10 provides the time spent by a processor to execute a row of supertasks. Suppose that, when the processor allocated to the row 0 finishes to be executed, there are m busy processors, i.e. there are m rows being executed. Therefore,

$$m(\frac{n^2}{k}+\lambda+\frac{n}{\sqrt{k}}\tau)-\lambda_r=\sqrt{k}(\frac{n^2}{k}+\lambda)$$

where $\lambda=\lambda_s+\lambda_r$. Let $X=\frac{n}{\sqrt{k}}$ and $Y=X^2+\lambda$. Then,

$$m=\left\lceil\frac{\sqrt{k}+\frac{\lambda_r}{Y}}{1+\frac{X}{Y}\tau}\right\rceil$$

In [24], Chochia *et al.* derived the makespan of a schedule for the diamond *DAG* partitioned into stripes with the same width considering the CLAUD model. In the evaluation, $\frac{n}{k'}$ lines of the diamond are arranged to form a stripe. Adjacent stripes are allocated to distinct processors and each stripe is partitioned in b regular blocks of $\frac{n^2}{bk'}$ tasks. The makespan for the stripe partitioning S_{Stripe} is then derived as:

$$\mathcal{M}(S_{Stripe}) = (\frac{n^2}{bk'}+\lambda_s+\lambda_r+\frac{n}{k'}\tau)(k'-1)+(\frac{n^2}{bk'}+\lambda_s+\lambda_r)b-(\lambda_s+\lambda_r)$$

The derived makespan $\mathcal{M}(S_{Stripe})$ is asymptotically optimal up to a factor of two in the case in which $\tau=0$ [24]. If $\tau=0$ or even the case in which $\lambda_s>\lambda_r\gg\tau$, the diamond *DAG* is partitioned into square blocks, each block of $\lambda_s+\lambda_r$ tasks. For values of b and k' such that $b=k'=\sqrt{k}$, S_{Stripe} is equivalent to S^{phys}_{boxes}.

9.4.2 Analysis of MSA on the diamond *DAG*

On the analysis of MSA, the $D_{n\times n}$, is used as the input *DAG*. Experiments are carried out and grouped in accordance with values of the communication parameters. The results are compared with the schedules $S^{virtual}_{boxes}$ and S^{phys}_{boxes}. For doing so, algorithms were implemented so that the makespan and schedule of both *boxes* partition techniques are precisely produced. Bear in mind that the objective of the comparison is not to show which is the best approach. The *boxes* partition is a very specific technique suitable for the diamond, while MSA is a scheduling approach for general *DAG*s. The comparison has as objective to show if MSA is able to provide a reasonable schedule for this class of *DAG*s.

Initially, the communication parameters, a fixed number of processors m_{fixed} and $D_{n\times n}$ are given as input to MSA. Then, the heuristics are executed varying the *clustering factor* γ as an attempt to define a good superdag. Let G^*_γ be the superdag produced by the first stage of MSA using γ and let γ_{best} be the γ-value that produces the best schedule. Also, let $S(G^*_\gamma)$ be the schedule produced by MSA for G^*_γ. Then, the superdag $G^*_{\gamma_{best}}$ is given as input to the second stage, which is executed by varying m, so that the smallest number of processors is provided. The simulation of the predictions is also carried out.

Results are shown in Table 9.1 were n^2 represents $D_{n\times n}$; $\lambda_s, \lambda_r, \tau$ are the communication parameters; γ_{best} is the *clustering factor* value that provides the best superdag; m_b is the number of processors necessary to provide the schedule with the smallest makespan; $\mathcal{M}(S_{MSA})$ the best makespan found by MSA; $k^{virtual}_{boxes}$ is the best number of processors for the *virtual boxes* partition described previously; $\beta^{virtual}_{boxes}$ is the number of tasks in each box for the given $k^{virtual}_{boxes}$ and finally, $\mathcal{M}(S^v)$ is the smallest makespan found for the *virtual boxes* partition. Correspondingly, k^{phy}_{boxes} and $\mathcal{M}(S^p)$ are related to the *physical boxes* partition.

9.4.2.1 Sending Costs

Initially, experiments were carried out for $D_{8\times8}$, considering that λ_s is the only significant parameter. For both values $\lambda_s = 4, 8$, the best makespan is smaller than $\mathcal{M}(S^v)$ and $\mathcal{M}(S^p)$; replication may result in less *sends* and it is only the *sending* overhead that is relevant. Observe that γ is very close to the *box* size. For the greater value $\lambda_s = 8$, the degree of parallelism is exploited by the replication of tasks, while the *boxes* partition executes all the application on one processor.

Figure 9.26 shows the results for a variety of *clustering factor* values and a fixed number of processors. The shape of the curve illustrates the fact that for

n^2	λ_s	λ_r	τ	γ_{best}	m_b	$\mathcal{M}(S)$	$k^{virtual}_{boxes}$	$\beta^{virtual}_{boxes}$	$\mathcal{M}(S^v)$	k^{phys}_{boxes}	$\mathcal{M}(S^p)$
64	4	0	0	14	6	47	4	16	56	16	52
64	8	0	0	14	3	54	1	64	64	16	64
64 †	0	0	4	11	11	40	4	16	41	—	—
64 †	0	0	8	23	3	49	4	16	49	—	—
256	2	0	0	0	18	119	64	4	102	64	88
256	5	0	0	35	18	148	16	16	157	64	130
256	10	0	0	35	18	167	16	16	192	16	172
256	0	4	0	6	27	162	64	4	140	64	116
256	0	4	0	20	25	153	64	4	140	64	116
256	0	8	0	53	6	194	16	16	184	16	160
256	0	15	0	53	6	222	4	64	237	16	202
256 †	0	0	2	0	16	63	64	4	67	—	—
256 †	0	0	8	14	32	121	16	16	115	—	—
256 †	0	0	20	27	31	168	4	64	169	—	—
256	4	1	0	20	25	153	16	16	152	64	130
256	8	3	0	35	18	184(171)	16	16	208	16	178
400 †	0	0	10	17	49	171	16	25	163	—	—
400 †	0	0	30	41	47	254	4	100	261	—	—
400 †	0	0	60	87	13	311	4	100	321	—	—
400 †	0	0	100	176	5	350	1	400	400	—	—
400	10	0	0	61	17	252	16	25	265	16	235
400	50	0	0	132	13	316	1	400	400	1	400
400	100	0	0	204	2	350	1	1	400	1	400
400	0	10	0	79	9	290	16	25	265	16	235
400	0	50	0	159	5	359	1	400	400	1	400
400	0	100	0	399	1	400	1	1	400	1	400
400	70	20	2	191	2	386	1	400	400	1	400
1024	10	4	1	128	26	660	64	16	586	64	464
1024	20	4	1	200	16	697	16	64	708	64	464
1024	50	4	1	215	16	745	4	256	938	16	796

†: unbundled messages
— : no unbundled version

Table 9.1: Results provided by MSA and *boxes* partition diamond *DAGs*.

$\gamma \ll \gamma_{best}$, the number of communication events can be high and for $\gamma \gg \gamma_{best}$ the degree of parallelism is very small. Two superdags were chosen: $G^*_{\gamma_{best}}$ and G^*_{γ}, where $\gamma > \gamma_{best}$ and also, where $\mathcal{M}(S(G^*_{\gamma}))$ is not much greater than $\mathcal{M}(S(G^*_{\gamma_{best}}))$. These two superdags are given to the second stage of MSA for various numbers of processors, so that it could be verified if there exists a certain number of processors in which a schedule with smaller makespan is produced. Figure 9.27 shows that this is not the case. For any value of m, γ_{best} produces a better schedule.

When dealing with a large *DAG* and relatively small overheads, such as $D_{16\times16}$ with $\lambda_s = 2$, the respective γ_{best} is very small, as would be expected. However, both *boxes* partitions produce a schedule with smaller makespan than that produced by MSA, even if a *clustering factor* greater than γ_{best} is given to the *replication* algorithm as an attempt to decrease the number of communication events.

On the other hand, when λ_s has a high value, MSA produces a better schedule than that specified by both *boxes* partition. This is a consequence of the minimisation of the number of *sending* events through replication of tasks. As seen in the above table, in the experiments on $D_{16\times16}$ with high λ_s, the size of the supertasks are practically double the size of each one of the *boxes*. In a system with high communication costs, MSA can produce good schedules by defining coarser superdags and minimising the number of *sending events*.

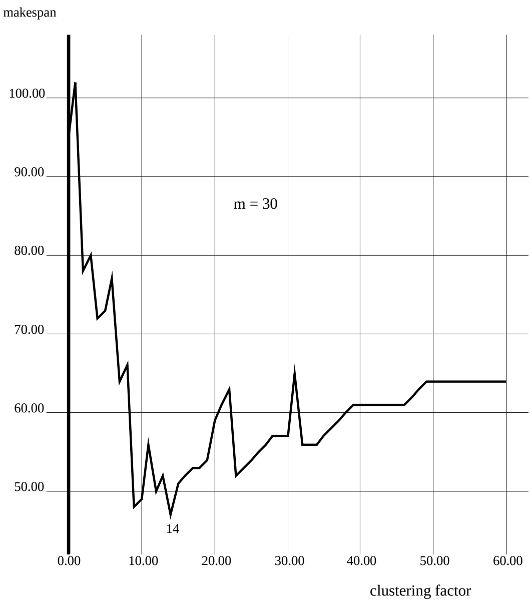


Figure 9.26: The $D_{8\times8}$ and $\gamma = 0, \ldots, 60$; $\lambda_s = 4$, $\lambda_r = 0$, $\tau = 0$ and $m = 30$.

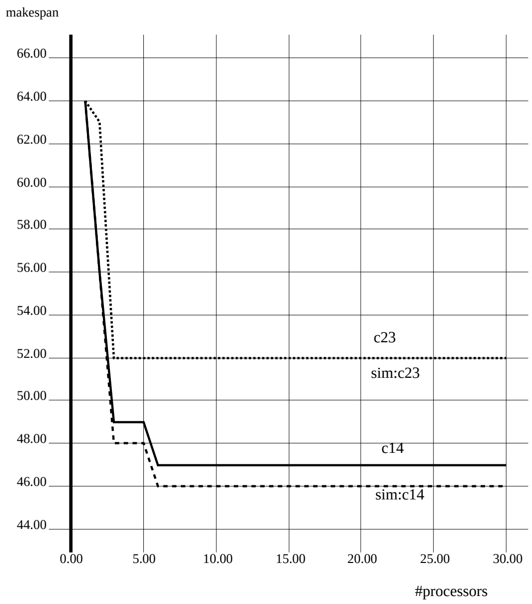


Figure 9.27: The $D_{8\times8}$ and $\gamma = 14$ and $\gamma = 23$; $\lambda_s = 4$, $\lambda_r = 0$, $\tau = 0$; $m = 1, \ldots, 30$.

Figure 9.28 shows the results of MSA for $D_{16\times16}$, $\lambda_s = 2$, $\lambda_r = 0$ and $\tau = 0$. The various curves in this Figure correspond to the different orders and characterisations of the rules of the *superdag* algorithm (first stage), on an attempt to evaluate and compare the schedules with the one provided by the *boxes* partition. However, as seen in the Figures 9.28 and 9.29, the best results are the one produced by $\gamma_{best} = 0$ (thus, no replication of tasks), using the following sequence of rules:

1) definition of the search space;
2) candidates which sends the largest messages;
3) candidates with minimal out degree;
4) candidates in the latest computation interval;
5) candidates with smallest topological order.

Figure 9.30 shows the results of MSA for $D_{16\times16}$ and the parameter values $\lambda_s = 5$, $\lambda_r = 0$ and $\tau = 0$. For the *clustering factors* $\gamma \ll \gamma_{best}$, the resulting

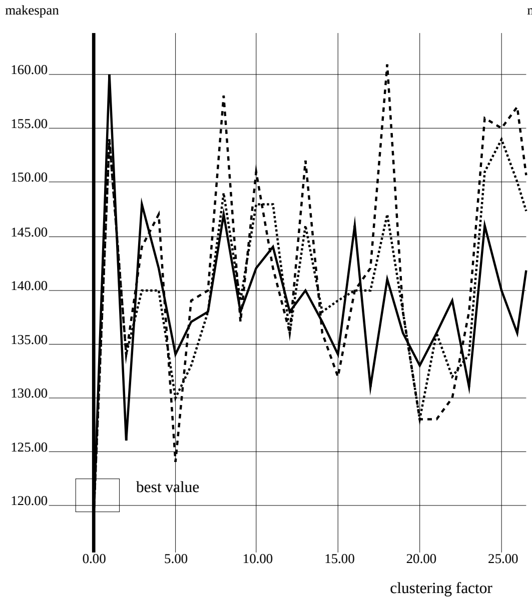


Figure 9.28: $D_{16\times16}$ and $\gamma = 0,\ldots,100$, $\lambda_s = 2$, $\lambda_r = 0$, $\tau = 0$, $m = 80$.

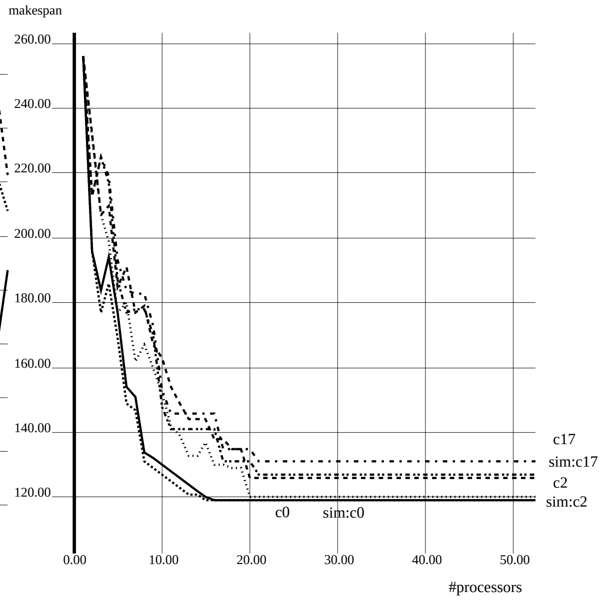


Figure 9.29: $D_{16\times16}$ and $\gamma = 0, 2, 17$, $\lambda_s = 2$, $\lambda_r = 0$,$\tau = 0$, $m = 1,\ldots,50$.

makespan is much greater than the one provided by $\gamma = \gamma_{best}$ and the curve becomes flatter from the best point. The curve in Figure 9.31 (prediction and time to completion of the simulations) and the numbers in Table 9.1 shows that the best number of processors for MSA is very close to $k^{virtual}_{boxes}$ but the number of tasks per supertask is greater than the number of tasks per *box*. The replication of tasks allows the increase of degree of parallelism without increasing the communication costs by executing coarser supertasks on each processor. For relatively high *sending overheads* (e.g. $\lambda_s = 10$), MSA produces better results than both *boxes* partitions.

9.4.2.2 Receiving Costs

We isolate the receiving overhead, although we are aware that this does not represent a real parallel machine. However, we would like to evaluate the performance of MSA when only *receiving events* are significant. The experiments for both $D_{16\times16}$ and $D_{20\times20}$ shows that both *boxes* partition can produce schedules with smaller makespan than that produced by MSA. However, when the value for λ_r increases, MSA may produce schedules with smaller makespan than that produced by the *virtual boxes* partition. Obviously, for higher λ_r values, γ_{best} also tends to be be high and consequently, less communication events are incurred. In the lines correspondent to $D_{16\times16}$, $\gamma_{best} = 6$, $\lambda_s = 0$, $\lambda_r = 4$, $\tau = 0$, we note that both *boxes* partition provide better results. In order to decrease

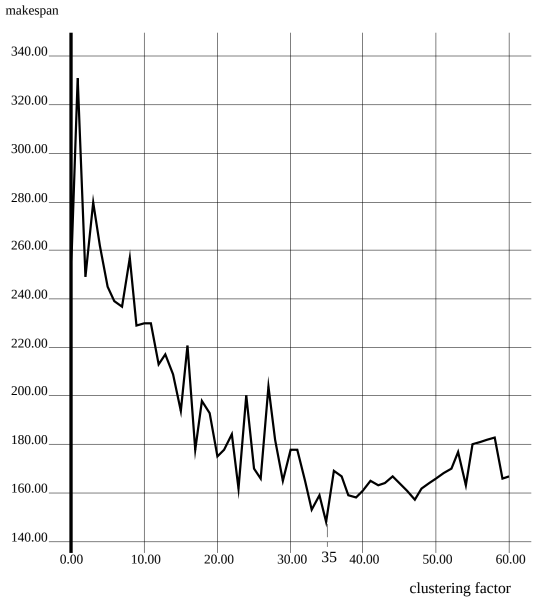


Figure 9.30: $D_{16\times16}$ and $\gamma = 0, \ldots, 60$, $\lambda_s = 5$, $\lambda_r = 0$, $\tau = 0$, $m = 100$.

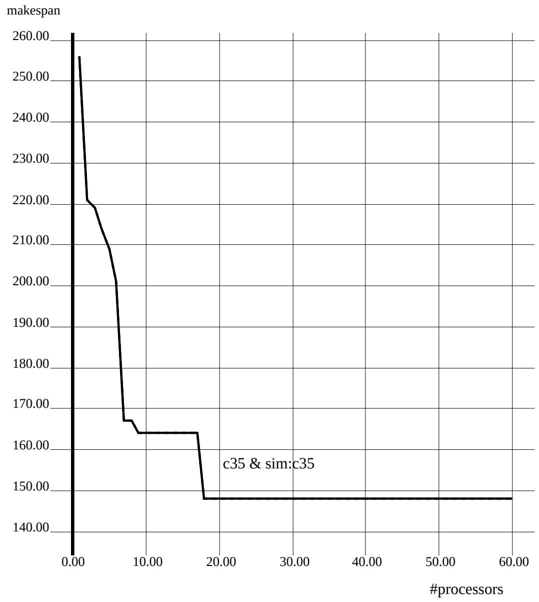


Figure 9.31: $D_{16\times16}$ and $\gamma = 35$, $\lambda_s = 5$, $\lambda_r = 0$,$\tau = 0$, $m = 1, \ldots, 60$.

the amount of communication events, we give to MSA value for γ greater than γ_{best}. For $\gamma = 20$, $\mathcal{M}(S(G^*_\gamma)) < \mathcal{M}(S(G^*_{\gamma_{best}}))$, but still, both *boxes* partition define better schedules. Nevertheless, for relatively high values of λ_r, MSA seems promising. For $D_{16\times16}$ and $\lambda_s = 0$, $\lambda_r = 15$, $\tau = 0$, MSA provides a schedule for $\gamma_{best} = 53$ such that $\mathcal{M}(S^p) < \mathcal{M}(S(G^*_{\gamma_{best}})) < \mathcal{M}(S^v)$. In Figures 9.32 and 9.33 we show the results produced by MSA. Values of γ greater than γ_{best} do not produced schedule with smaller makespan than $S(G^*_{\gamma_{best}})$.

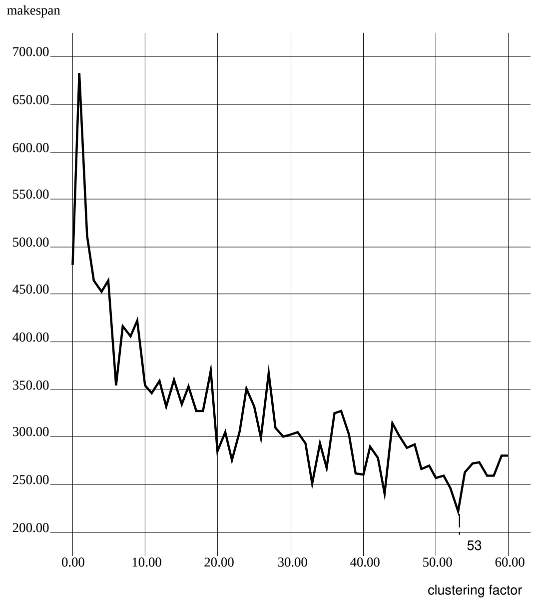


Figure 9.32: $D_{16\times16}$ and $\gamma = 0,\ldots,60$, $\lambda_s = 0$, $\lambda_r = 15$, $\tau = 0$, $m = 100$.

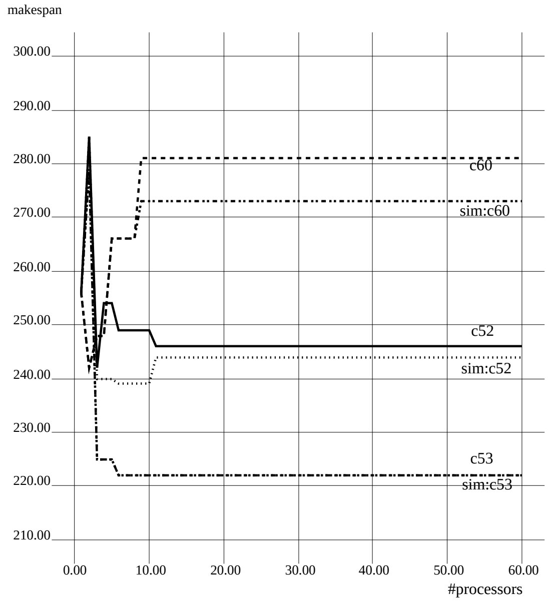


Figure 9.33: $D_{16\times16}$ and $\gamma = 50,....$, $\lambda_s = 0$, $\lambda_r = 15$,$\tau = 0$, $m = 1,\ldots,60$.

Therefore, for high receiving overheads, γ_{best} should be used so that MSA can provide a good schedule.

9.4.2.3 Delay cost

When the *delay* model is considered, the proper set of rules in the *superdag* algorithm must be applied. Also, the *unbundled-messages* procedure is performed after the new allocation produced by the *remapping* algorithm is established. In order to do a fair comparison with the *boxes* partition, a version of the *boxes* strategy in which messages are *unbundled* is also implemented. We have implemented this version only for the *virtual boxes* partition.

The makespan of the schedules of $D_{8\times8}$ considering γ_{best} for $\tau = 4$ and $\tau = 8$ are very close to the best ones produced by the *virtual boxes* partition. Figure 9.34 shows that for $\tau = 8$, the best makespan can be achieved either for $\gamma = 10$ or $\gamma = 23$. In Figure 9.35, when $\gamma = 23$, a smaller number of processors is sufficient to achieve the best makespan than in the case $\gamma = 10$.

In order to evaluate the impact of small and large delays (which can also correspond to applications in which tasks send short or long messages, respectively) relative to the size of the input *DAG*, $D_{16\times16}$ is scheduled under the *delay* model where τ is associated with the values 2, 8 and 20. For very low delay costs, the *boxes* partition may provide better results (see also Figures 9.36 and 9.37). However, for larger delays, MSA produces good results considering

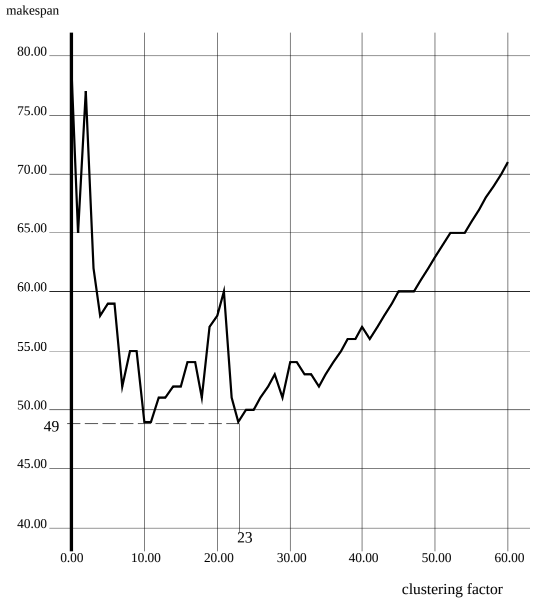


Figure 9.34: $D_{8\times 8}$ and $\gamma = 0, \ldots, 60$, $\lambda_s = 0$, $\lambda_r = 0$, $\tau = 8$, $m = 50$.

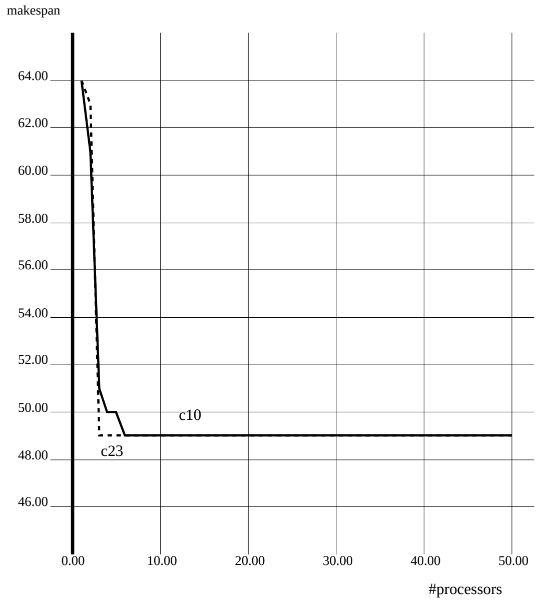


Figure 9.35: $D_{8\times 8}$ and $\gamma = 10, 23$ $\lambda_s = 0$, $\lambda_r = 0$, $\tau = 8$, $m = 1, \ldots, 50$.

γ_{best} as the *clustering factor* and the best number of processors. Note that the best *clustering factor* is not necessary equal to τ. Although in the *delay* model communication can be totally overlapped with useful computation of tasks, the number of *computational* intervals produced for $\gamma = \tau$ may be greater than the number of *computational* intervals when $\gamma > \tau$. The *clustering factor* γ_{best} depends not only in the communication costs but also in the *structure of the input DAG*.

9.4.2.4 All of the communication parameters

In the case in which all the communication parameters are not negligible, mainly when the *sending* overhead is higher than the other parameters, MSA produces better results than both *boxes* partition.

In Table 9.1, in the lines relative to $D_{32\times 32}$, we see clearly the benefits of MSA when high overheads characterise the target architecture. For smaller communication costs, both *boxes* partition provide a better schedule, although MSA does not specify a schedule with makespan much higher than $\mathcal{M}(S^v)$ and $\mathcal{M}(S^p)$.

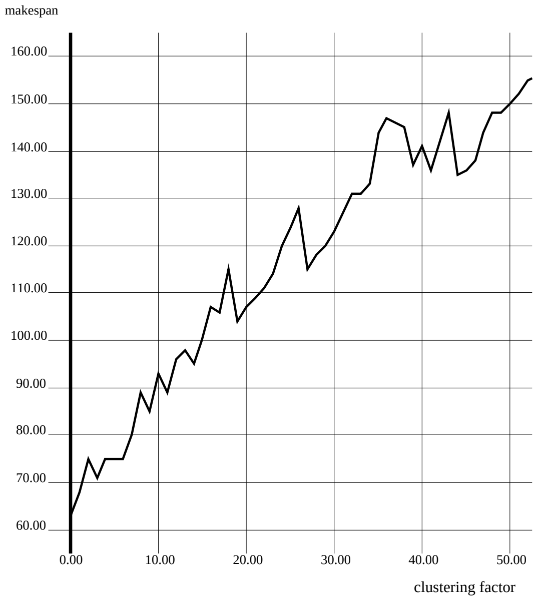


Figure 9.36: The 16×16 diamond *DAG*: $\gamma = 0, \ldots, 50$, $\lambda_s = 0$, $\lambda_r = 0$, $\tau = 2$, $m = 50$.

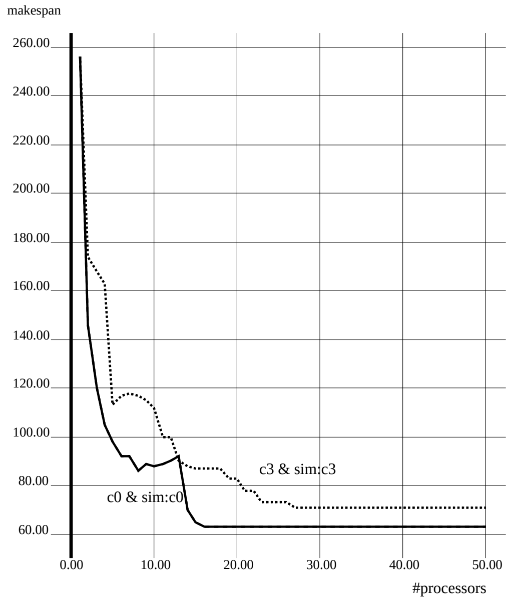


Figure 9.37: The 16×16 diamond *DAG*: $\gamma = 0, 3$, $\lambda_s = 0$, $\lambda_r = 0$,$\tau = 2$, $m = 1, \ldots, 50$.

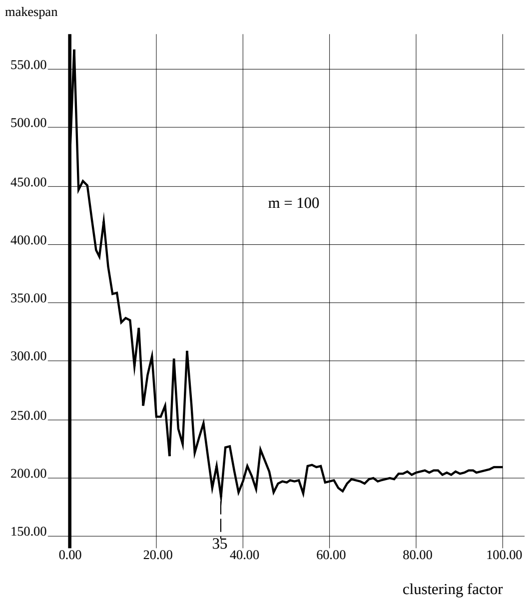


Figure 9.38: The 16×16 diamond *DAG*: $\gamma = 0, \ldots, 100$, $\lambda_s = 8$, $\lambda_r = 3$, $\tau = 0$, $m = 100$.

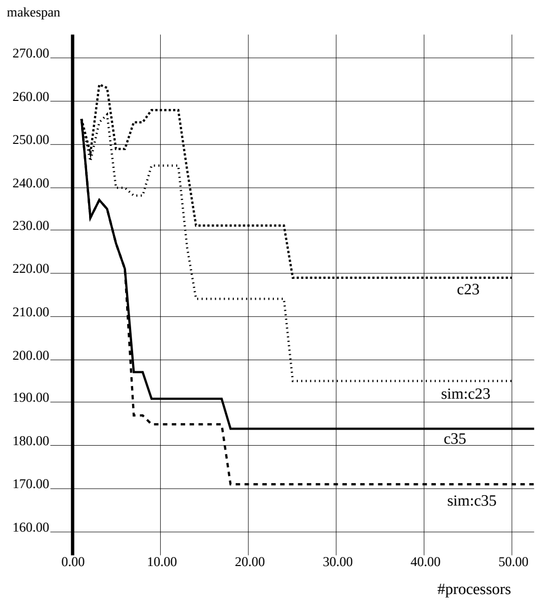


Figure 9.39: The 16×16 diamond *DAG*: $\gamma = 23, 35$, $\lambda_s = 8$, $\lambda_r = 3$,$\tau = 0$, $m = 1, \ldots, 50$.

9.4.3 Comparing with ETF and MH

Both the MH and the ETF algorithms were executed for $D_{20 \times 20}$ considering the *delay* model, where the delay values were $\tau = 30$, $\tau = 60$ and $\tau = 100$. As expected, the best results produced by both heuristics correspond to the schedule of the input *DAG* on one processor. Also, for the same values of the other communication parameters shown in Table 9.1, considering $D_{20\times20}$, the best schedules produced by MH and ETF are when $D_{20\times20}$ is executed on one processor, apart from the case $\lambda_s = 10$, $\lambda_r = 10$ and $\tau = 0$, in which the schedule produced by ETF has makespan equal to 265 and two processors are used.

9.4.4 Concluding remarks

The experimental evaluation shows the benefits of task replication mainly when the communication parameters have high values. The *multi-stage approach* performs well in *DAG*s like the diamond, producing schedules with makespan better (whenever the communication costs are high) or close to the one produced by the *virtual* and *physical boxes* partitions. The objective of the comparison with these partitions was to verify the performance of MSA for *DAG*s such as the diamond.

The comparison gives some *hints* on what can be done to improve MSA. When the communication costs are not very high (relative to the size of the input diamond *DAG*), we observed that MSA may provide a schedule with makespan greater than both *boxes* partitions. The factor that is causing the increase is the amount of replication. Therefore, a mechanism that controls this degree, replicating tasks only when it is really necessary is a topic to be examined in the future.

We conclude that MSA is very promising when comparing with heuristics such as ETF and MH, which have poor performance when communication costs are high.

Chapter 10

Conclusions, Discussions and Further Work

10.1 Concluding remarks

This thesis has highlighted the relevance of *communication event* tasks and the importance of modelling clusters of tasks of a *DAG* and their relationship using the *superdag* concept. The *communication events* represent overheads which are incurred on processors in the form of a *CPU penalty* due to sending and receiving of messages. These kinds of overheads must be taken into account since they are significant in many parallel machines. A scheduling strategy that is proposed to produce good schedules considering a wide range of parallel computer cannot neglect these overheads anymore. When the *communication events* are considered, communication is not totally overlapped with computation of tasks. However, a great number of scheduling techniques model the startup overhead as part of the communication delay.

The *multi-stage approach*, MSA, is proposed where, not only the delay on communication is considered when formulating a scheduling, but also, *communication event* tasks are identified. In the attempt to decrease the number of such *communication events* that occurred on the processors along the critical path, messages are bundled into larger ones. This approach is efficient when the overheads are much greater that the delay, i.e. $\lambda_s \gg \tau$ and $\lambda_r \gg \tau$. In order to bundle messages, tasks are clustered. The clusters of tasks and their relationship is represented by the superdag.

MSA incorporates *task replication*, an advantageous technique mainly when communication costs are very high. In the case in which overheads have critical values, the replication of tasks is a way to decrease the number of communication events. However, it is not always the case that task replication minimises the number of *communication events*. The “structure” of the input *DAG* also dictates the degree of replication that leads to the specification of a good schedule. The *replication* algorithm in the first stage of MSA does not replicate tasks arbitrarily, but considers the adjacency among tasks so that an initial attempt to minimise the number of *communication events* is carried out. The *replication* algorithm uses two priorities to cluster tasks: the first one is the e_l-value of each task, which is the latest time that a task can start without increasing the makespan and the second, the number of immediate successors already in the cluster. The definition of the e_l-function increases the number of candidate tasks that can be clustered together. The methods used to cluster tasks is the first attempt to decrease the number of communication events. After that, it is necessary to carefully define the relationship amongst the clusters, which represents the *communication events*.

In MSA, a series of mechanisms is implemented with the objective to produce a good schedule of an application onto a parallel computer in which the overheads due to sending and receiving are very high. The *superdag* algorithm, the second heuristic in the first stage of MSA, implements these mechanisms or *rules*. As a result, a reasonable superdag for the underlying target architectural model is specified. One of the main objectives is to define supertasks with low degrees, mainly those supertasks which are in the critical path of the superdag.

Nevertheless, the range of architectural models to which MSA can be applied is not restricted to those models with overhead much higher than the delay. The rules proposed in this thesis can be ordered and parameterised in such a way that good schedules can also be provided for those models in which, for example, the delay in the only significant parameter, i.e. the *delay* model. The different orders and parameterisations are necessary because of the different nature of the two main communication costs considered in this thesis: the *blocking* overheads and the *non-blocking* delay. Another important fact is that it is not only the architectural parameters that guide MSA, but also, characteristics of the input application. Therefore, both application and target architecture must analysed so that the parameters of the heuristics are properly assigned and the rules are well parameterised and ordered.

Although the delay or latency on communications has been an issue widely evaluated and considered not only by new scheduling strategies but also, by hardware designers whose efforts led to communication developments and enhancements, this parameter is still a significant characteristic in parallel computation. Even if delay is small in comparison to other communication parameters, the size of messages exchanged between tasks (or supertasks) may be considered as a characteristic that "delays" the time to completion of the application.

The definition of *computational* and *communication* intervals is an opportunity to increase the degree of parallelism and also, facilitates a more precise prediction of the number of *communication events*, especially when the bundling of messages is made necessary. Even when the *delay* is the only significant parameter, the definition of the intervals leads to very promising results, because the degree of parallelism can be better maximised. Note that our main objective in this thesis when defining *computational* and *communication* intervals is not to specify *synchronisation points*. With the definition of such intervals, the second stage of MSA is implemented and the identification of potential parallel supertasks allows the possibility to "rearrange" them onto physical processors more efficiently.

Other benefits provided by these intervals can be pointed out. For example, in general, list scheduling algorithms perform decisions in accordance with the local state of the schedule process. These strategies produce near-optimal schedules which depends on the *granularity* of the application. The optimality of ETF, for example, depends on the size of messages and on the delay on communications. The identification of well-defined *computation intervals* in MSA is an attempt to make decisions less associated with the local state of the scheduling process.

Brent's Principal is very relevant and promising when the maximisation of parallelism is an objective to be achieved. Although the Principle in its essence does not account for the allocation and scheduling of tasks on the target network, the *remapping* algorithm uses Brent's Principle in order to maximise the degree of parallelism. Obviously, this degree is decreased by other characteristics such as the precedence relationship between tasks and the communication costs. The *remapping* algorithm implements mechanisms which minimise the number of *communication events* and the delay on communications. The *common-immediate-successor* mechanism is intended to decrease the number of *communication events* by allocating adjacent supertasks to the same processor, whenever possible. The *look-at-others* mechanism aims to "force" the allocation of the most communicating-intensive adjacent supertasks to the same processor and therefore, the communication delay is also minimised. In the $free\_allocation$ procedure, the mechanism *successor-same-processor* is also an attempt to decrease both the number of *communication events* and to allocate to the same processor communicating supertasks which send maximal-length message. Basically, these remapping mechanisms analyse the structure of the input superdag (or *DAG*) produced by the first stage of MSA and both the number of communication events and the delay cost are minimised.

The evaluation of well known regular *DAG*s and also, irregular *DAG*s, shows the benefits provided by MSA. When evaluating different partitioning techniques on *Trees*, it was verified that depending on the significance of the communication parameters, different partitions produce optimal schedules. Tuning the parameters of MSA, our strategy can produce these same schedules. The schedule of these particular class of *DAG*s showed that the best arrangement to *group* tasks together depends not only on the architectural parameters, but also, in the characteristics of the input *DAG*. Establishing the conclusions of the different partitions considering the Full Binary Tree, experiments were performed. Firstly, a comparison between the different partitions for different models is carried out. The benefits of using the *layer* partition for the *delay* model and the *subtree* partition when both overhead parameters are significant, are verified. Also, better results are produced by MSA for the Full Binary Tree than those produced by ETF and MH, mainly when communication costs are high. However, for this type of *DAG*, ETF and MH results are not so unreasonable. The same cannot be said when scheduling the diamond *DAG*. The MSA results are compared with the *boxes* partition for different target models and both results are correlated. This is mainly due to the manner in which tasks are clustered by the *replication* algorithm. On the other hand, ETF and MH do not produce reasonable results for this kind of *DAG*. For relatively high communication costs, both heuristics schedule the diamond *DAG*s on one processor.

Experimental evaluations using irregular *DAG*s showed the overall performance of MSA. The definition of the *intervals* was advantageous in the sense that it was "easier" to predict a proper schedule of the *communication event* tasks. The experimental results showed that the curves representing MSA predictions and the time to completion of the respective simulations are closely correlated. The MSA schedules can be considered an upper bound for the time to completion of the simulations. In the case of the other heuristics, whenever the number of communication events is extremely high and also the amount of concurrency is high, the predictions do not correspond well with the reality. This can be easily seen by the comparison between the ETF and MH predictions and respective simulations. The discrepancy between the two heuristics predictions and the simulations is explained mainly by the following facts. These heuristics minimise the makespan by scheduling the tasks of the application in such a way that tasks computation is *totally* overlapped with communication. Another crucial problem is that messages are not bundled and to make things worse, the startup overhead is incorporated to the delay. When giving the resulting schedules to a parallel machine in which the startup overhead is very high and it is part of the processor job, a high number of *communication* events are incurred. Therefore, these heuristics (mainly MH, since it defines startup overhead when sending messages) are not realistic. Through the bundling of messages, MSA avoids the specification of such unreasonable results. Through the versatility of the rules, MSA can produce good schedules for different models.

The results produced by MSA in the various experiments show that the schedule with the smallest makespan is usually achieved with a reasonable number of processors. This is due to task replication which, in attempting to minimise communication costs, might mean more processors are necessary in order to produce a good schedule. This effect was analysed in [125] which concluded that to achieve the optimal makespan using the replication strategy of Papadimitriou and Yannakakis, an exponential number of processors is necessary. There exists in the literature, some replication strategies which follow a similar line of thought and always assume sufficient numbers of processors are available to execute the "good" schedule they produce.

10.2 The versatile nature

When discussing the models of parallel computation, we have mentioned some of the many models that cannot be considered realistic, since they do not take into account many important issues of the parallel machines. However these models cannot be discarded, since they are important tools on the complexity evaluation of parallel algorithms. We know how difficult it is to tackle a problem considering many different aspects of parallel machines. The usefulness of such simplistic models becomes clear when it is necessary to analyse specific aspects of parallel computer. Nevertheless, the need of a representative model for a wide range of parallel machines led to the definition of the general models of parallel computation. Their main objective is specification of an easier interface between computational and programming models. Although these models are supposed to represent a parallel environment more accurately, it is hard to specify many of the characteristics of parallel computers without hindering the usability of the general model. Therefore, only the parameters that can represent various parallel machines are used. Still, the lack of the definition of certain important characteristics can be identified in some of these models. This lack is clearly observed when extensions to these models are proposed.

Similarly in the scheduling field, many strategies were proposed taken into account specific characteristics of parallel machines and applications. These strategies, although not general nor realistic, cannot be ignored since they are regarded as important techniques. On the other hand, the specification of a single efficient heuristic that takes into account a whole set of important parameters can be a hard, maybe impossible, task. This difficulty is stressed when the parameters are of different nature, such as the communication parameters in CLAUD.

This work is not the first one in which, when devising a scheduling strategy where different parameters can be considered, the need to specify a series of heuristics has risen. When devising MSA, it was very difficult to consider all of the parameters at once so that an efficient result was provided. Consequently, a series of transformations were implemented, corresponding to the final form of MSA.

Another interesting point was identified while devising MSA. Each one of the many *rules* and *mechanisms* in each of the heuristics of the first and second stages, were devised knowing that a *local solution* should be avoided. The assumptions considered when defining the heuristics of MSA were an attempt to predict the behaviour of the scheduling process based on current information. As already discussed throughout the specification of the rules, the heuristics are greedy algorithms. When using MSA on general applications, the solution is not necessarily the optimal one (we know that we are dealing with a NP-complete problem and that MSA is an approximation heuristic). A solution far from the optimal may be achieved mainly when some assumptions taken into consideration do not occur in future stages of the scheduling process.

This discussion conducts our thoughts to the construction of an "intelligent" heuristic with a more precise view of the future, so that the rules could be applied in a more efficient way. The versatility of the heuristics is an important mechanism to be used when the scheduling process depends on many characteristics of the target architecture. Also, the structure of the input *DAG* is important. Therefore, rather than statically selecting the heuristics in the beginning of the scheduling process, the selection would be performed during the scheduling process, in accordance with the *global view* of the intelligent heuristics.

The actual version of MSA can be considered as a general scheduling technique, since through the initial analysis of the target architecture and a possible analysis of the input *DAG*, a proper set of heuristics is applied. The novelty in MSA is not necessarily identified in each of the heuristics, but in the way that the rules were combined. With the development of parallel computations and the enhancement of their performance, different characteristics are identified as *bottlenecks* along the evolution of the machines. A versatile technique is an attempt to follow the changes caused by the evolution of parallel computation.

10.3 Future work

The many aspects of parallel computation give us the opportunity to suggest some points to be studied. On the proposal of MSA, a heuristic that deals with realistic issues as specified in the CLAUD model, we came across onto matters that need still to be improved and also, aspects that are relevant due to the evolution of parallel computation.

Can we automate the detection of the "best" *clustering factor* γ, considering the input *DAG* and the communication parameters without having to execute MSA for various values of γ ? The evaluation of the various replication approaches discussed in the brief survey in Section 3.3 is proposed in order to incorporate mechanisms in our *replication* algorithm, so that a good prediction of the best value for γ can be derived. The majority of these heuristics use a cost function in order to guide their scheduling decisions. As future work, the adaptation of others replication techniques, e.g. that of Lee and Hurson [81, 82], is proposed as an attempt to define more automatically the γ parameter. However, these techniques should be adapted to the superdag concept, so that the *bundling of messages* is exploited.

The versatility of MSA needs to be more fully exploited. Evaluations on the *Balanced Subset Choice* Rule need to be performed so that the proper ranges of the communication parameter values are established. In this way, the applicability of this rule can be automatically defined.

Although communication protocols are being developed so that the problem of topologies is hidden, the mapping of tasks on processors is still an issue to be evaluated and its importance analysed, considering different topologies. In the current version of MSA, the notion of topology is not incorporated, since MSA only produces schedules for fully connected network. Therefore, all the characteristics of the CLAUD model are fully tackled. The question on whether to incorporate the notion of topology in the second stage or to propose a *third* stage in which this issue is considered, is proposed as future work. Nevertheless, as a first step, an evaluation of current and future communication protocols should be done, so that more realistic assumptions are made when devising a new *mapping* methodology.

The study of the applicability of our method to general models of parallel computation such as the LogP model is suggested. The LogP model is still being evaluated and extensions are being proposed [6]. The g parameter defined in LogP is suggested to be incorporated to the test-bed environment and a proper definition of rules considering such communication characteristic is a work yet to be analysed. Intuitively, each supertask should have a small out degree if such parameter is considered significant. Also, the performance analysis of the superdag concept considering models like BSP is relevant and it is a work already initiated in [7]. However, a question may arise: should replication of tasks be applied when synchronisation costs are significant?

Congestion is also a relevant problem. In order to analyse congestion when scheduling an application on a parallel computer, one should have a *global view* of the system. Scheduling algorithms that perform local decisions may produce local optimal solutions. The analysis done on list scheduling algorithms shows the difficulties and pitfalls of a one-stage strategy. The same is applied to the occurrence of *communication events*. In the clustering of tasks and in the definition their dependencies, there may be situations in which the number of communication events could be still minimised. As future work, the investigation of the possibility to incorporate strategies that considers a *global view* of the system being scheduled is suggested, so that global decisions are taken rather than local ones. The existence of *communication event* tasks should be considered and also, the problem of *congestion* should be taken into account.

When investigating the state-of-the-art of parallel machines, we identified an emergent class of parallel computers in which a hierarchical architecture is implemented, such as the Convex machines. The HPRAM model [58, 59] considers this hierarchical parallelism, although communication costs like overheads are not incorporated. In order to exploit the hierarchical parallelism implicit in applications, scheduling techniques which handle such a characteristic should be devised.

Bibliography

[1] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedulers for parallel processing systems. *Comm. ACM*, 17(12):685–690, 1974.

[2] T. Agerwala, J.L. Martin, J.H. Mirza, D.C. Sadler, D.M. Dias, and M. Snir. SP2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995. Scalable Parallel Computing.

[3] A. Aggarwal, A.K. Chandra, and M. Snir. On communication latency in PRAM computations. Technical Report RE RC 14973, IBM T.J. Watson Research Center, Yorktown Heights, NY, Sep 1989.

[4] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.

[5] I. Ahmad and Y-K Kwok. A new approach to scheduling parallel programs using task duplication. In K.C. Tai, editor, *International Conference on Parallel Processing*, volume 2, pages 47–51, Aug 1994.

[6] A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model - one step closer towards a realistic model for parallel computation. *7th Annual Symposium on Parallel Algorithms and Architectures (SPAA'95)*, 1995. Also as TRCS95-09 - Dept Comp.Science - Univ. St. Barbara.

[7] R.J. Anderson, P. Beame, and W. Ruzzo. Low overhead parallel schedule for task graphs. *SPAA*, pages 66–75, 1990.

[8] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):23–39, February 1995.

[9] F.D. Anger, J-J Hwang, and Y-C Chow. Scheduling with sufficient loosely coupled processors. *Journal of Parallel and Distributed Computing*, 9:87–92, 1990.

[10] G. Astfalk. A fair view of MPP parallelism. *European Convex Users Conference (Spain)*, Oct 1993.

[11] G. Astfalk. Parallel programming on the Convex MPP. Technical report, Convex Computer Corporation, PO Box 833851 - Richardson, TX 75083-3851, 1994.

[12] V. Bala, J. Bruck, R Bryant, R. Cypher, P. de Jong, P. Elustindo, D. Frye, A. Ho, C-T Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4), April 1994.

[13] E. Bampis, F. Guinand, and D. Trystram. Some models for scheduling parallel programs with communication delays. To appear in *Discrete Applied Mathematics*.

[14] E. Bampis, J-C. Konig, and D. Trystram. Optimal parallel execution of complete binary trees and grids into most popular interconnection networks. *Theoretical Computer Science*, 147:1–18, 1995.

[15] E. Bartson, J. Cownie, and McLaren. M. Message passing on meiko CS-2. *Parallel Computing*, 20(4), April 1994.

[16] J. Baxter and J.H. Patel. The LAST algorithm: A heuristic based static task allocation algorithm. In *Proc. Intl. Conf. Parallel Comput.*, volume 2, pages 217–222, 1989.

[17] D. Bilardi and F.P. Preparata. Horizons of parallel computing. In A. Bensoussan and J.P. Verjus, editors, *in Future Tendencies in Computer Science, Control, and Applied Mathematics (International Conference on the Occasion of the 25th Anniversary of INRIA)*, pages 155–174, December 1992.

[18] C. Boeres, G. Chochia, and P. Thanisch. On the scope of applicability of the ETF algorithm. In Afonso Ferreira and Jose Rolim, editors, *Parallel Algorithms for Irregularly Structured Problems (Second International Workshop, IRREGULAR'95)*, Lecture Notes in Computer Science, LNCS 980, pages 159–164, Lyon, France, September 1995. Springer.

[19] S.H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, C-30(3):207–214, 1981.

[20] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.

[21] T. Brewer. A highly scalable system utilizing up to 128 PA-RISC processors. Technical report, Convex Computer Corporation, 1995.

[22] T. Casavant and J. Kuhl. A taxonomy of scheduling in general purpose distributed computing systems. *IEEE Trans. Software Engrg.*, SE-14(2):141–154, 1988.

[23] H.B. Chen, B. Shirazi, K. Kavi, and A.R. Hurson. Static Scheduling Using Linear Clustering with Task Duplication. *Proceedings of ISCA International Conference on Parallel and Distributed Computing Systems*, pages 285–290, 1993.

[24] G. Chochia, C. Boeres, M. Norman, and P. Thanisch. Analysis of multicomputer schedules in a cost and latency model of communication. In *Proceedings of the Third Workshop on Abstract Machine Models for Parallel and Distributed Computing*, Devonshire Hall, Leeds, April 1996. Parallel Processing Specialist Group of the BCS and the Leeds University School of Computer Studies, IOS press.

[25] Y. C. Chung and S. Ranka. Applications and performance analysis of a compile time optimization approach for list scheduling. In *Proceedings of Supercomputing' 92*, pages 512–521, Nov 1992.

[26] E.G. Coffman Jr., editor. *Computer and Job Shop Scheduling Theory*. John Wiley, New York, 1976.

[27] E.G. Coffman Jr. and R.L. Graham. Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213, 1972.

[28] R. Cole and O. Zajicek. The APRAM - The Rounds Complexity Measure and the Explicit Costs of Synchronization. Technical Report TR-539, Courant Institute of Mathematical Sciences, Dep. of Comp. Sci. - NY University, Jan. 1991.

[29] J.Y. Colin and P. Chrétienne. C.P.M. scheduling with small communication delays and task duplication. *Operations Research*, 39(4):680–684, July 1991.

[30] T.H. Cormen, C.E. Leiserson, and Rivest R.L. *Introduction to Algorithms*. The MIT Elect. Eng. and Comp. Science Series. MIT Press, 1990.

[31] M. Cosnard and A. Ferreira. Designing parallel non numerical algorithms. *Parallel Processing Letters*, 1(2):103–111, 1991.

[32] M. Cosnard and D. Trystram. *Parallel Algorithms and Architectures*. Int. Thomson Computer Press, 1995.

[33] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems : Concepts and design*. International Computer Science Series. Addison-Wesley, 2nd ed. edition, 1994. Electronic data processing – Distributed processing Distributed computer systems.

[34] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[35] S. Darbha and D.P. Agrawal. SDBS: A task duplication based optimal scheduling algorithm. *Proceedings of the Scalable High-Performance Computing Conference*, pages 756–763, May 1994.

[36] S. Darbha and D.P. Agrawal. A task deuplication based optimal scheduling algorithm for variable execution time tasks. In K.C. Tai, editor, *International Conference on Parallel Processing*, volume 2, pages 52–56, Aug 1994.

[37] J.J. Dongarra. Performance of various computers using standard linear equations software. TR CS-89-85, Computer Science Dept. - University of Tennessee, Knoxville, TN 37996, December 1991.

[38] T.H. Dunigan. Performance of a second generation hypercube. TR ORNL/TM-10881, National Lab, Oak Ridge, November 1988.

[39] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. Technical Report UCB/CSD92/62, Computer Science Division - EECS, University of California, Berkeley, CA 94720, March 1992.

[40] H El-Rewini. *Task Partitioning and Scheduling on Arbitrary Parallel Processing Systems*. PhD thesis, Dept. of Comp. Science - Oregon State University, 1990.

[41] H. El-Rewini and T.G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Dist. Comput.*, 9:138–153, 1990.

[42] H. El-Rewini, T.G. Lewis, and H.A. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, New Jersey, 07632, 1994.

[43] Message Passing Forum. MPI: a Message Passing Interface. Technical report, University of Tennessee, 1995.

[44] L. Gao, A. L. Rosenberg, and R.K. Sitaraman. Optimal architecture-independent scheduling of fine-grain tree-sweep computations. In *Proc. of the 7th IEEE Symp. on Parallel and distributed Processing*, pages 620–629, 1995.

[45] A. Gerasoulis, Jiao J., and T. Yang. A multistage approach to scheduling task graphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 22:81–103, 1995.

[46] A. Gerasoulis, S. Venugopol, and T. Yang. Clustering task graphs for message passing architectures. In *The International Conference on Supercomputing*, pages 447–456, Amsterdam, The Netherlands, June 1990.

[47] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.

[48] D. Ghosal, A. Mukherjee, R. Thurimella, and Y. Yesha. Mapping tasks trees onto linear arrays. University of Denver. Manuscript submitted for publication, 1995.

[49] A. Gibbons. An Introduction to Distributed Memory Models of Parallel Computation. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, chapter 10, pages 197–226. Cambrigde University Press, 1993.

[50] P.B. Gibbons. A more practical PRAM model. In *Proc. 1st Symp. on Parallel Algorithms and Architectures*, 1989.

[51] A. Gottlieb. An overview of the NYU ultracomputer project. In J.J. Dongarra, editor, *Experimental Parallel Computing Architectures*, pages 25–95. North Holland, 1987.

[52] A. Gottlieb. Ultra III: Implementing a scalable shared-memory multiprocessors. Technical report, CS Dept - New York University, 1989.

[53] A. Gottlieb, L. Rudolph, and B. Lubachesky. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM TOPLAS*, 5(2):164–189, 1983.

[54] R.L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.

[55] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimisation and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–236, 1979.

[56] S.E. Hambrusch and A. Khokhar. C^3: An architecture-independent model for coarse-grained parallel machines. *The 6th IEEE Symposium on Parallel and Distributed Processing*, pages 544–551, 1994.

[57] T. Heywood and C. Leopold. Models of parallelism. Technical Report CSR-28-93, Dept. of Comp. Sci. - University of Edinburgh, July 1993.

[58] T. Heywood and S. Ranka. A Pratical Hierarchical Model of Parallel Computation: Binary Tree and FFT Graph Algorithms. *Journal of Parallel and Distributed Computing*, 3:233–249, 1992.

[59] T. Heywood and S. Ranka. A Pratical Hierarchical Model of Parallel Computation: The Model. *Journal of Parallel and Distributed Computing*, 3:212–232, 1992.

[60] M.D. Hill. What is scalability? *Computer Architecture News*, 18(4):18–21, Dec 1990.

[61] J.W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proc. 13th ACM Symposium on the Theory of Computing*, pages 326–333, May 1981.

[62] J.A. Hoogeven, S.L. van de Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocation. Technical Report Report BS-R9211, CWI, Netherlands, June 1992.

[63] F.W. Howell. Reverse profiling. In Innes Jelly, Ian Gorton, and Peter Croll, editors, *Software Engineering for Parallel and Distributed Systems*, pages 244–255. Chapman and Hall on behalf of IFIP, March 1996.

[64] T.C. Hu. Parallel sequencing and assembly line problems. *Oper Res.*, 9:841–848, 1961.

[65] J-J. Hwang, Y-C. Chow, F.D. Anger, and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[66] K. Hwang. *Advanced Computer Architecture - Parallelism, Scalability, Programmability*. McGraw Hill, 1993.

[67] Intel. Personal communication, 1991.

[68] K. Johnson and A. Agarwal. The impact of communication locality on large-scale multiprocessor performance. Technical Report MIT/LCS/TM-463, MIT Laboratory for Computer Science, February 1992.

[69] S.L. Johnsson. Communication in network architectures. In R. Suaya and G. Birtwhistle, editors, *VLSI and Parallel Computation*, pages 223–378. Morgan Kaufmann, Palolto, CA., 1990.

[70] H. Jung, L. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of DAGs with communication delays. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 254–264, 1989.

[71] R. M. Karp, A. Sahay, E.E. Santos, and K.E. Schauser. Optimal Broadcast and Summation in the LogP Model. In *Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, Velen, Germany, June30 - July 2 1993. ACM, ACM Press.

[72] R.M. Karp and V. Ramachandran. Parallel Algorithms for Shared Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 870–941. Elsevier Science Publishers, 1990.

[73] K. Keeton, T. Anderson, and D. Patterson. LogP quantified: The case for low-overhead local area networks. In *Proceedings of Hot Interconnects III: A Symposium on High Performance Interconnects*, Stanford, CA, August 1995.

[74] A.A. Khan, C.L. McCreary, and M.S. Jones. Comparison of multiprocessor scheduling heuristics. In K.C. Tai, editor, *Proc. of the Eighth International Parallel Processing*, volume II, pages 243–250, Cancun, Mexico, April 1994. IEEE Computer Society Press - ACM SIGARCH.

[75] S.J. Kim and J.C. Browne. A general approach to mapping of parallel computations upon multiprocessor architectures. In *The 3rd International Conference on Parallel Processing*, pages 1–8, 1988.

[76] R.K. Koeninger, M. Furtney, and M. Walker. A shared memory MPP for CRAY research. *Digital Technical Journal*, 6(2), Spring 1994. Cray Research papers.

[77] B. Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Processing Systems*. PhD thesis, Dept. of Electrical and Computer Eng., Oregon State University - Corvallis, 1987.

[78] B Kruatrachue and T. Lewis. Grain size determination for parallel programming. *IEEE Software*, pages 23–32, January January 1988.

[79] C.P. Kruskal, L. Rudolph, and M Snir. Efficient synchronization on multiprocessors with shared memory. *ACM TOPLAS*, 10(4):579–601, Oct 1988.

[80] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, July 1978.

[81] B. Lee and A.R. Hurson. A strategy for scheduling partially ordered program graphs onto multicomputers. *Parallel Processing Letters*, 5(4):575–586, 1995. Special Issue on Partitioning and Scheduling in Parallel Systems.

[82] B. Lee and A.R. Hurson. A strategy for scheduling partially ordered program graphs onto multicomputers. In *Proceedings of the 28th Annual Hawaii International Conference on System Science*, pages 133–142, 1995.

[83] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufmann, San Mateo, CA., 1992.

[84] C.E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, C34(10):892–900, 1985.

[85] C.E. Leiserson *et al.* The network architecture of the Connection Machine CM-5. In *Proc. ACM Symp. Parallel Algorithms and Architectures*, San Diego, CA, 1992.

[86] T.G. Lewis. Piece-wise scheduling of composite graphs onto distributed memory parallel computers. Technical Report TR91-60-15, Comp. Science Dept - Oregon State University, Corvallis, OR 97331-3202, 1991.

[87] V. Lo, V. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, and J. Telle. Mapping divide-and-conquer algorithms to parallel architectures. Technical Report CIS-TR-89-19, Dept of Computer and Information Science, University of Oregon, Eugene, Oregon 97403, January 1990.

[88] W. Lowe and W. Zimmermann. On finding optimal clusterings of task graphs. In *Aizu International Symposium on Parallel Algorithm and Architecture Synthesis*, pages 241–247. IEEE Computer Society Press, 1995.

[89] W. Lowe and W. Zimmermann. Upper time bounds for executing PRAM-programs on the LogP-machine. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 41–50, 1995.

[90] P. Manneback, J. Quin, and G. Libert. Performance models of MICCG algorithm on distributed memory MIMD computers. In W. Joosen and E. Milgrom, editors, *Parallel Computing: from Theory to Sound Practice*, pages 52–59. IOS Press, 1992.

[91] S. Manoharan. *Task Assignment in Parallel Processor Systems*. PhD thesis, Department of Computer Science - Univ. of Edinburgh, 1992.

[92] C. Martel, R. Subramonian, and A. Park. Asynchronous PRAM are (almost) as good as synchronous PRAMs. In *Proceedings of the 31st Annual Symposium of the Foundations of Computer Science*, pages 590–599, 1990.

[93] W. F. McColl. General purpose parallel computing. Technical report, Programming Research Group, Oxford University, April 1992.

[94] W.F. McColl. General Purpose Parallel Computing. In Alan Gibbons and Paul Spirakis, editors, *Lectures on Parallel Computation*, chapter 13, pages 337–391. Cambridge University Press, 1993.

[95] W.F. McColl. BSP programming. In G Blelloch, M Chandy, and S Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms, Princeton, May 9-11, 1994*. American Mathematical Society, 1994.

[96] W.F. McColl. Scalable computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments. LNCS Volume 1000*, pages 46–61. Springer-Verlag, 1995.

[97] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel programming. *Comm. ACM*, 32(9):1073–2078, 1989.

[98] M.E. McCreary and M.E McArdle. A comparison of task partitioning techniques. Technical Report TR CSE-91-14, Dept. of Comp. Science and Eng. - Auburn University, Auburn, AL 36849-5347, Dec 1991.

[99] M.E. McCreary and Y. Shu. Shortest path evaluation for hierarchical grain aggregation. Technical Report TR CSE-93-03, Dept. of Comp. Science and Eng. - Auburn University, Auburn, AL 36849-5347, 1993.

[100] M.E. McCreary, J. Thompson, H. Gill, T. Smith, and Y. Shu. Partitioning and scheduling using graph decomposition. Technical Report TR CSE-93-06, Dept. of Comp. Science and Eng. - Auburn University, Auburn, AL 36849-5347, 1993.

[101] N. Nishimura. Asynchronous shared memory parallel computation. *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 76–84, 1990.

[102] M.G. Norman, G Chochia, P Thanisch, and E Issman. Predicting the performance of the diamond DAG computation. Technical Report EPCC-TR-92-07, Edinburgh Parallel Computing Centre, 1992.

[103] M.G. Norman and P. Thanisch. Models of machines and computations for mapping in multicomputers. *ACM Computing Surveys*, 25(3):263–302, 1993.

[104] M.G. Norman, P. Thanisch, and K.-F. Chang. Partitioning DAG computations: A cautionary note. In W. Joosen and E. Milgrom, editors, *Parallel Computing: from Theory to Sound Practice*, pages 360–364. IOS Press, 1992.

[105] C.H. Papadimitriou and J.D. Ullman. A communication-time tradeoff. *SIAM J. Comput.*, 16(4):639–646, 1987.

[106] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19:322–328, 1990.

[107] B. Parhami. SIMD Machines: Do they have a significant future? In *The Fifth Symp. on Frontiers of Massively Parallel Computation*, McLean, VA, Feb 1995. Sponsored by IEEE Computer Society. Report on a Panem Discussion at Frontiers 95.

[108] G. Pathak and D.P. Agrawal. Task division and multicomputer system. In *Proc. 5th Int. Conf. on Distributed Computing System*, Denver, CO, 1985.

[109] D.A. Patterson. Observations on massively parallel processors: A case for a new theoretical model: LogP. In *The DIMACS Workshop of Models, Architectures, and Technologies for Parallel Computation*, Rutgers, NJ, USA, Sept 1993. Invited Talk.

[110] D.A. Patterson and J.L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub. Inc., 1990.

[111] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design - the hardware/software interface*. Morgan Kaufmann Pub. Inc., San Mateo - CA 94403, 1994.

[112] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4):463–480, April 1994.

[113] C.D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell Mass., 1988.

[114] M.J. Quinn. *Parallel Computing Theory and Practice*. McGraw-Hill, 1994.

[115] Fortune S. and J. Wyllie. Parallelism in random access machines. In *Proc. of 10th Annual ACM Symposium on Theory of Computing*, pages 114–118. ACM Press, 1978.

[116] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London, 1989.

[117] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. *Communications of the ACM*, pages 17–26, 1986.

[118] M. Schmidt-Voigt. Efficient parallel communication with the nCUBE 2S processor. *Parallel Computing*, 20(4):509–530, April 1994.

[119] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *J. Parallel Dist. Comput.*, 10:222–232, 1990.

[120] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–157, 1991.

[121] H. Stone. *High-Performance Computer Architecture*. Electrical and Computer Engineering. Addison-Wesley, 3rd edition, 1993.

[122] M.N.S. Swamy and K. Thulasiram. *Graphs, Networks and Algorithms*. John Wiley and Sons, 1981.

[123] P. Thanisch and M.G. Norman. Bounds beyond which there are no performance gains from adding processors to scheduling models. Technical Report EPCC-TR-93-06, Edinburgh Parallel Computing Centre, 1990.

[124] P Thanisch, M.G. Norman, C. Boeres, and S. Pelagatti. Exponential processor requirements for optimal schedules in architectures with locality. *Proc. of the Workshop on General Purpose Parallel Computing*, December 1993. British Computer Society - Parallel Processing Specialist Group(BCS PPSG) - London.

[125] P Thanisch, M.G. Norman, C. Boeres, and S. Pelagatti. Locality in Scheduling Models of Parallel Computation. In Karsten M. Decker and René M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 265 – 274. Birkhäuser Verlag, Basel, July 1994.

[126] R. Thurimella and Y. Yesha. A scheduling principle for precedence graphs with communication delay. *J. of Computer and Software Engineering*, 2(2):165–176, 1994.

[127] J.D. Ullman. NP-complete scheduling problems. *J. of Computer and System Sciences*, 10:384–393, 1975.

[128] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, 1990.

[129] L.G. Valiant. General Purpose Parallel Architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 944–971. Elsevier Science Publishers, 1990.

[130] B. Veltman, B.J. Lageweg, and J.K. Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16(2-3):173–182, 1990.

[131] S. Venugopal. Linear clustering of task graphs for local memory systems. Private Communication.

[132] R.J. Wilson and J.J. Watkins. *GRAPHS, An Introductory Approach*. John Wiley and Sons, Inc., 1990.

[133] M. Wu and D. Gajski. A programming aid for hypercube architectures. *Journal of Supercomputing*, 2(3):349–372, Nov. 1988.

[134] J. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Comp. Sci. Dept. – Cornell University, Ithaca, NY, 1979.

[135] T. Yang. *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, Graduate School - The State University of New Jersey, New Brunswick - Rutgers, 1993. Ph.D Thesis.

[136] T. Yang and A. Gerasoulis. PYRROS: Static scheduling and code generation for message passing multiprocessors. In *Proc. of 6th ACM International Conference on Supercomputing*, pages 428–437, Washington D.C., July 1992.

[137] T. Yang and A. Gerasoulis. List scheduling with and without communication delay. *Parallel Computing*, 19:1321–1344, 1993.

[138] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[139] W. H. Yu. *LU Decomposition on a Multiprocessor System with Communication Delay*. PhD thesis, Dept. of Elec. Edng. and Comp. Science, Univ. of California, Berkeley, 1984.

[140] W. Zimmermann and H. Kumm. On the implementation of the virtual shared memory. In *Programming Models for Massively Parallel Processing*, pages 172–178, Berlin, Germany, Sept 20-23 1993. IEEE Comp. Soc. Press.

Appendix A

List of symbols

$G = (V, E)$	application represented by a *DAG*
$\mathcal{P}$	a set of m physical processors
λ_s	sending overhead
λ_r	receiving overhead
τ	delay on communication
$h(p_i, p_j)$	number of hops between processors p_i and p_j
$\varphi(p_i, p_j)$	$\tau * h(p_i, p_j)$
$pred(v_i)$	immediate predecessors of v_i
$succ(v_i)$	immediate successors of v_i
$ancestors(v_i)$	the list of all ancestors of task v_i
$m(v_i)$	execution cost of v_i
$w((v_i, v_j))$	weight associated with the edge (v_i, v_j)
$path(v_{i_1}, v_{i_k})$	the path between any two tasks $v_{i_1}, v_{i_k} \in V$
$spath(v_{i_1}, v_{i_k})$	one of the shortest paths between two tasks v_{i_1}, v_{i_k}
$\mathcal{T}(v_i)$	the topological order of a task v_i
$proc(v_i)$	the processor in which v_i is uniquely allocated
$processor(v_i)$	the list of processors in which v_i is replicated
$\sigma_{(v_i, v_j, p)}$	sending event task allocated to p, associated with (v_i, v_j)
$\rho_{(v_i, v_j, p')}$	receiving event task allocated to p', associated with (v_i, v_j)
$s(v_i, p)$	the start time of v_i on processor p
$s(v_i)$	the start time of v_i (when there is no replication)
$f(v_i, p)$	the finish time of v_i on p
$f(v_i)$	the finish time of v_i when there is no replication
S or $S(G)$	the set of scheduling tuples of *G*
$\mathcal{M}(S)$ or $\mathcal{M}(S(G))$	the makespan of the schedule of *G*
$(v_i, p_j, t) \in S$	task v_i is scheduled on processor p_j at time *t*
$idg(v_i)$	in degree of task v_i
$odg(v_i)$	out degree of task v_i
$\mathcal{W}(G)$	width of the *DAG G*
$C_{(v_i, v_j)}$	$w((v_i, v_j)) \cdot \varphi(proc(v_i), proc(v_j))$
$CMax_{(v_i, v_j)}$	$\leq C_{(v_i, v_j)} + odg(v_i)\lambda_s + idg(v_j)\lambda_r$

$c = \langle v_{i_1}, \ldots, v_{i_{n_c}} \rangle$	a cluster of tasks
$Clusters(v_i, S)$	the set of all clusters where v_i is replicated
v_{v_i, p_k}	a copy of v_i on processor p_k
$e_{PY}(v_i)$	the e-value of v_i as defined in [106]
$e_l(v_i)$	the e_l-value of v_i defined in the *replication* algorithm
$elist_l(v_i)$	the ancestors of v_i listed in their decreasing order of e_l-values
$V_{cluster}$	the set of clusters produced by the *replication* algorithm
$\mathcal{I}(c)$	$\{u_k \in V \ : \ (u_k, v_{i_j}) \in E \ \wedge \ u_k \notin c \ \wedge \ v_{i_j} \in c\}$ — all the immediate predecessors of tasks in c, which are outside c
$useless(c)$	if $TRUE$, c is an immediate predecessor cluster
$\mathcal{B}$	$\left\lceil \frac{\mathcal{M}(S)-(2\gamma-1)}{2\gamma+1} \right\rceil$ — the number of bands with parallel supertasks
β	a band in the schedule S
$start(\beta)$	the start time of β
$finish(\beta)$	the finish time of band β
$\mathcal{C}(\beta)$	the list of parallel supertasks executed during β
$\eta(\beta)$	$\lceil \frac{\mathcal{C}(\beta)}{m} \rceil$ — number of supertasks reallocated to a processor
$slist(p_j, \beta)$	all supertasks reallocated to processor p_j during β
$\Gamma(p_j, \beta)$	immediate predecessor of all supertasks reallocated to p_j during $\beta - 1$
$lproc(\beta)$	all processors already used during $\beta - 1$