THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e. g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Collection Skeletons - Declarative Abstractions for Data Collections

THE UNIVERSITY *of* EDINBURGH

Zhibo Li

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2024

Abstract

Modern programming languages provide programmers with rich abstractions for data collections as part of their standard libraries, e.g., Containers in the C++ STL, the Java Collections Framework, or the Scala Collections API. Typically, these collections frameworks are organised as hierarchies that provide programmers with common abstract data types (ADTs) like lists, queues, and stacks. While convenient, this approach introduces problems which ultimately affect application performance due to users overspecifying collection data types limiting implementation flexibility. On the other hand, with the development of parallel computing, there are increasingly parallel architectures that provide parallel speedup for diverse applications such as big data and artificial intelligence, however, there is still a substantial gap between the parallel hardware and writing parallel code. Additionally, the overspecification issue makes it harder for programmers, and particularly non-professional programmers to develop parallel programs. This work develops Collection Skeletons which provide a novel, declarative approach to data collections. Using our framework, programmers explicitly select properties for their collections, thereby truly decoupling specification from implementation. By making collection properties explicit, immediate benefits materialise in the forms of reduced risk of overspecification and increased implementation flexibility. A prototype library of the declarative abstractions for collections is presented in this thesis and shows that benchmark applications rewritten to use Collection Skeletons incur little or no overhead for sequential circumstances. In fact, performance improvements have been reported across most of the 17 benchmarks resulting from the use of Collection Skeletons before trying to parallelize those benchmarks, while also enhancing performance portability across three different hardware platforms. Additionally, by extending the Collection Skeletons towards parallelization, this work shows that Collection Skeletons help shielding the application developer from parallel implementation details, either by encapsulating implicit parallelism or through explicit properties that capture the requirements of parallel algorithmic skeletons. The Collection Skeletons help the programmers write better (parallel) code by providing a property-based declaration for data collections while creating parallel opportunities for library designers who can transparently improve the overall program performance without requiring much effort from the programmers.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

The following refereed papers have been published during this PhD course. These form the basis for parts of this thesis as indicated.

Björn Franke, **Zhibo Li**, Magnus Morton, and Michel Steuwer
"Collection Skeletons: Declarative Abstractions for Data Collections " (Awarded **The Best Research Paper**)
in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, Auckland, New Zealand, 2022.
— This publication forms the basis for Chapter 4

Björn Franke, **Zhibo Li**, Magnus Morton, and Michel Steuwer
"Collection Skeletons: Declarative Abstractions for Data Collections "
The Journal of Systems and Software, Volume 213, July 2024, 112042.
— This work forms the basis for Chapter 5 and Chapter 6

(*Zhibo Li*)

Lay Summary

Modern programming languages offer programmers easy-to-use tools to interact with data collections. These tools are typically organised in complicated hierarchies which hide implementational details from the programmers. However, such hierarchies tend to make the programmers difficult to choose a suitable data container from. Many developers also struggle with writing parallel programs, even though there are many tools available for modern parallel computing. This is mainly because they are not familiar with these parallel programming frameworks. Together with the difficulty with data collections, developing parallel programs can be challenging for many programmers, especially for those who are non-professional.

To address this problem, we propose a novel method named "Collection Skeletons". This approach simplifies data collections by removing the complex hierarchies. Instead, programmers specify what they need from the data collection without knowing the specific data structure. We focus on two main types of requirements: what the data collection should mean (semantic properties), and what actions can be performed on the data collection (interface properties).

Our experiments show that this new method is efficient and can improve performance in many cases. To make parallel programming easier, we introduce two strategies: Implicit Parallelism and Explicit Parallelism. With Implicit Parallelism, developers can write code as if it is sequential, and the proposed method handles the parallel processing automatically in background. Explicit Parallelism, on the other hand, requires developers to use specific algorithmic skeletons to fulfill their logic. Our evaluations indicate that both strategies, along with Collection Skeletons, provide a simple yet powerful way to write parallel programs, improving both ease of use and performance.

Acknowledgements

Firstly, I extend my deepest gratitude to my supervisor, Björn Franke, for his invaluable support and guidance over the past few years. His optimism, rigor and remarkable research experience have been fundamental in exploring our topic. Beyond academic guidance, I am profoundly grateful for his assistance in various aspects of my daily life.

I also wish to express my sincere thanks to Murray Cole, whose work on algorithmic skeletons laid the groundwork for my research. His tireless mentorship and insightful feedback have been immensely beneficial to my work. Additionally, I am indebted to Michel Steuwer and Magnus Morton for their unwavering support, particularly during the hard times of the pandemic. Their contributions were instrumental in the great success of my first paper.

My colleagues at ICSA deserve special mention; I am thankful for Christos Vasiladiotis's sincere support during the last few years. I am also thankful for Alexandr Maramzin's valuable assistance during my initial year. My heartfelt thanks to Haoyu, Jackson, Pablo, David, Bo, Celeste, and others whose camaraderie has enriched my PhD life.

I am particularly grateful to my supervisor at the South China University of Technology, Kejing He, whose generous assistance provided a significant impetus to my research and career. The endless support and understanding of my parents have been my pillar of strength.

Lastly, I owe a profound debt of gratitude to my wife, Frida, whose unwavering love and encouragement have been a source of joy and confidence in both my work and life.

Table of Contents

1 Introduction **1**
1.1 Solution: Programming with Collection Abstractions 6
1.2 Motivating Examples 8
1.2.1 Problem: Overspecifying / Suboptimal Choices on the Data Structure 8
1.2.2 Problem: Overspecifying the Control Structure 11
1.2.3 Problem: Over-integration of Data and Control Abstraction . . 14
1.3 Contributions 15
1.4 Organisation of the Thesis 16

2 Background **17**
2.1 Data Structures and Abstract Data Types 17
2.1.1 Introduction to Data Structures 17
2.1.2 Abstract Data Types 18
2.2 Data Structure Libraries and Hierarchies 19
2.2.1 Scala Collection Library 19
2.2.2 C++ Standard Template Library 20
2.2.3 Boost Library 22
2.3 Parallel Computing 22
2.3.1 Implicit Parallelism 23
2.3.2 Explicit Parallelism 26
2.4 Algorithmic Skeletons 27
2.4.1 Introduction to Algorithmic Skeletons 27
2.4.2 Simple Data-centric Algorithmic Skeletons 29
2.4.3 Task-based Algorithmic Skeletons 31
2.4.4 Other Data-centric Algorithmic Skeletons 35
2.5 Skeletons-based Parallel Computing 37

2.5.1 Algorithmic Skeletons in Standard Libraries of Modern Programming Languages . . . 38
2.6 Chapter Summary . . . 40

3 Related Work **41**
3.1 Specification of Data Collections . . . 41
3.2 Collection Hierarchies . . . 43
3.3 Parallelisation in the Presence of Pointer-Based Data Structures . . . 44
3.3.1 Parallel Programming and Data Dependency . . . 45
3.3.2 Decoupled Software Pipelining . . . 46
3.4 Data Structure Detection & Shape Analysis . . . 48
3.5 Data Structure Replacement . . . 49
3.6 Common Implementations of Algorithmic Skeletons . . . 51
3.7 Chapter Summary . . . 54

4 Declarative Abstractions for Data Collections **55**
4.1 Introduction . . . 55
4.2 Overview of Collection Skeletons . . . 57
4.3 Properties Identification . . . 62
4.4 Properties for Collection Skeletons . . . 66
4.4.1 Semantic Properties . . . 68
4.4.2 Interface Properties . . . 68
4.4.3 Hybrid Properties . . . 69
4.4.4 Non-functional Properties . . . 70
4.5 Design Principles of the Prototype Library . . . 71
4.5.1 API of Collection Skeletons . . . 72
4.5.2 Concrete Implementation Decision —the Flexibility . . . 74
4.5.3 Design of the MSPM Algorithm . . . 77
4.5.4 Rules of Properties and API Design . . . 79
4.5.5 Implementation of the Pattern Matching Algorithm . . . 80
4.5.6 Discussion on Alternative Designs . . . 83
4.6 Evaluation . . . 83
4.6.1 Experimental Results . . . 86
4.7 Chapter Summary . . . 92

5 Exploiting Implicit Parallelism with Collection Skeletons 93
5.1 Introduction to Implicit Parallelism 94
5.2 Providing Implicit Parallelism with Collection Skeletons 95
5.2.1 Integration with Collection Skeletons 97
5.2.2 Performance Boost by Implicit Parallelism 98
5.2.3 Beyond Implicit Parallelism 99
5.3 Implementation —Prototype Library Extension 100
5.3.1 Concurrent Data Structures in the Collection Skeletons Framework 100
5.3.2 Thread Safety 102
5.3.3 Beyond the Preparation for Parallelisation 104
5.4 Evaluation 105
5.4.1 Evaluation on the Concurrent Data Structures 105
5.4.2 Evaluation with the Introduction of OpenMP 107
5.5 Chapter Summary 109

6 Explicit Parallelism —Integration of Collection & Algorithmic Skeletons 111
6.1 Introduction 111
6.2 Integration of Both Types of Skeletons 113
6.2.1 Challenges on Exposing Properties for Algorithmic Skeletons 114
6.2.2 Properties of Collections That Relate to the Application of an Algorithmic Skeleton 115
6.2.3 Property Check for Collection Skeletons with Algorithmic Skeletons 117
6.2.4 Property Inference 118
6.3 Data-centric Algorithmic Skeletons 120
6.3.1 Map 120
6.3.2 Reduce 122
6.3.3 Filter 122
6.3.4 Zip 123
6.3.5 Implementation of the Data-centric Algorithmic Skeletons . . 124
6.4 Advanced Algorithmic Skeletons 127
6.4.1 Stencil 127
6.4.2 Wavefront 131
6.4.3 Divide-and-conquer 134

6.5 Evaluation . . . 136
6.5.1 Evaluation of the Extension of Data-centric Algorithm Skeletons 136
6.5.2 Evaluation on Stencil Skeleton . . . 138
6.6 Chapter Summary . . . 142

7 Conclusion 144
7.1 Summary . . . 144
7.2 Reflection . . . 145
7.3 Conclusion . . . 145
7.4 Future Work . . . 146

Bibliography 147

Chapter 1

Introduction

Parallel Computing has become increasingly important in current research and application in many areas [6, 49, 54]. With the advancement of the multi-core CPU and GPUs, parallel computing has become the de facto configuration for many application fields including big data analysis and Artificial Intelligence [133, 112, 110]. Emerging accelerators such as FPGAs provide domain specific computational speedup as an integral part of parallel computing, even further improving the computational capability for more problem domains [17, 120]. Novel hardware for parallel computing has been seen in recent years. However, the corresponding software for parallel computing has been developing slowly [3]. For example, despite the powerful speedup by NVIDIA's GPUs, application developers are still required to program with CUDA in C programming language; they still need to know every detail of the memory model to harness the performance of NVIDIA's GPU. They need to explicitly manage the data storage and transfer as well as memory allocations on both device side and host side, leaving them unable to only focus on implementing their algorithms. The cost of parallel programming thus substantially hinders the broader application of parallel programming [116, 115, 109, 40], and will cause high cost in software engineering [77, 99, 122]. Even frameworks such as oneAPI, focusing on providing abstract interfaces for parallel programming, still have substantial learning curves and even performance tradeoffs [78, 29].

The mismatch of parallel software and parallel hardware will ultimately cause a problem —the programmers find it difficult to write parallel code [68]. Empirically, we classify programmers based on their experience on programming for a specific domain —1. **Specialised programmers** are good at programming within that specific topic, e.g., they are experts for a library or a programming language, or experienced on parallel programming. Specialised programmers only account for a small number of

overall programmers. However, they develop high quality software including tools and libraries used by others. 2. **Non-specialised programmers** are not professional programmers and of limited ability with programming for a specific topic. Non-specialised programmers correspond to the major population of programmers, they are not good at developing software for a specific topic, e.g., they are not good at writing parallel programs. For non-specialised programmers regarding parallel programming, the numerous parallel computing platforms and their numerous corresponding frameworks are difficult to follow given they might have been already struggling at interacting with the programming languages. For any of the parallel framework mentioned above, e.g., if the non-specialised programmers for parallel programming are writing parallel code with SYCL [41], they need to overcome the sharp learning curve from the SYCL standard and the implementational details of one of the implementations such as oneAPI. In fact, despite the advance of parallel computing frameworks in recent years, the most used ones are still challenging to non-specialised programmers, and the programmability for programmers has not developed well. For example, the non-specialised programmers need to have a detailed understanding of the memory and threads of an NVIDIA GPU if they want to write parallel code with CUDA; or if they want to write multi-threaded code with C++, they need to understand the synchronisation mechanism implemented by atomics, locks and the interactions between the threads and the programming API of pthreads or similar thread libraries introduced in modern C++ [14].

In this thesis, we refer to programmers who struggle to interact with data collections or write parallel programs as non-specialized programmers, unless otherwise specified. For non-specialised programmers, it is difficult to write parallel code to harness the parallelism of the emerging hardware. This situation can be improved by a transition and simplification of the workload from the programmers side —they can write sequential code first and then transform the sequential code into parallel code. In fact, with the most popular parallel computing frameworks such as OpenMP and MPI [42, 139], the programmers can first write the sequential version of their code and then insert the corresponding compiler directives or reorganise the code for distributed communication, to achieve a parallel version of the original code. Furthermore, there are automatic parallelism backed by specific compilers such as DPC++ [24]. However, this two step approach does not save the programmers from writing parallel code —they can not even write good sequential code, the transformation could make the whole thing worse. While this two-step process can transform sequential code into parallel code, it does not inherently improve the efficiency of poorly written sequential code. The goal of writing

parallel code is to enhance performance, not merely to parallelize for its own sake.

Another persistent challenge for writing parallel program is that different hardware requires different software/programming languages implementations, hindering program portability across different architectures. For example, it is expensive to migrate a CUDA program to OpenCL [105] to make it workable on an AMD GPU; or it is similarly difficult to migrate a CUDA program to oneAPI to harness the parallel performance from Intel's novel GPUs [134], even though the latter is designed to be a unified compatible standard for heterogeneous parallel computing. In fact, a unified standard such as SYCL could do little to existing and legacy code written in CUDA —the legacy code will still be there and be functional, not to mention programming with SYCL still needs intensive interaction with details such as memory allocation/data transferring.

Even worse, at the system software layer, the compilers do not provide comparable support by generating parallelised code. For example, when programming with C++/C programming languages, the user allocates an array at runtime. The compiler can hardly do optimisation on this array as it would never know its information except as a block of memory. The circumstances become even more challenging when working with 2-dimensional arrays and higher-dimensional arrays. Declaring a dynamic array with a pointer has been a long popular operation in C++/C programming especially in legacy code and given situation when standard library is not available, either because of the hardware limitation or because the programmers do not even know the information of the array prior to runtime. There are many similar situations as of mentioned above where usage of some data structures will lead to non-optimizable structures for compilers. However, with the example above, if we know the size of the array in advance of runtime, we can store the data into a bracket-declared array or an array from a third-party library; Then, the compiler can properly optimise the array along with the operations associated with the array. In fact, by simply altering the data structure, the compiler can do more optimisation for the programmer. With this idea, however, how can we fulfil the potential of compilers without adding additional complexity for non-specialised programmers?

The root of both the portability and performance problems is that the programmer has to **overspecify** the code including both details of **choice of data structures and control structure**. What works well on one system or in one use case might not be suitable or optimal for another. Even if programmers begin with sequential code and then port it to parallel platforms, this approach does not guarantee the easy writing of effective parallel code. Not only non-specialised programmers, but also specialised programmers,

tend to overspecify data structures during programming tasks. For example, when a singly-linked list is most suitable for the problem domain, the programmer chooses an array instead, or vice versa. By developing the program with more suitable data structures, the runtime computational performance and memory performance can be improved; moreover, there can be more opportunities for parallelism to be exposed.

While newly emerging parallel computing libraries aim to address classic programming challenges, non-specialised programmers often find these solutions add complexity rather than reducing it. For example, SkePU is a skeleton focused parallel computing framework for heterogeneous platforms [48]. To work with SkePU, the programmers need to write algorithmic skeletons based programs and store the data into the containers provided by SkePU. While SkePU helps solve some problems of the programmability, new issues such as scaling arise that would still hinder the programmers writing parallel code. With SkePU, the expressibility is limited as the programmers have to pick up from the limited number of containers provided by SkePU, while for those data structures such as self-defined data types that are not provided, they can not write parallel code with SkePU. Furthermore, a series of algorithmic skeletons can not cover all the problem domains especially those requiring complex logic [98], which further limits the expressiveness of SkePU. For other frameworks such as Taskflow [70], though they help solve specific domains of problems, the expressiveness is still limited.

To help solve the data collection misspecification issue by programmers, a novel interaction mechanism with data collection is needed. Let us consider how the overspecification of the data collection arises —the programmers have a bunch of data and need to store the data into a data collection; then, they develop a series of algorithms to process the data from the data collection. During the picking of the data collection, the programmers have several choices —for example, if the programmer is working with C++, then they can choose either 1. implement the data structure themselves; 2. pick one from C++ STL or Boost; 3. find a third-party data structure library. Each of the strategies requires a lot of understanding on the data structures —the programmer needs to know the implementation details of the data structure that they are going to use. However, for many non-specialised programmers, it is not practical to thoroughly understand the data structures, not to mention the details such as the memory and computational properties of these data structures. Even for specialised programmers who have a good understanding of data structures and their usage, it is still non-trivial to pick the most appropriate one —thus overspecification will happen.

Let us reconsider data and the containers/collections that store the data without

concerning ourselves with how the compiler optimises them or how to harness the multi-core performance through parallel programming. Collections of data items are central to many fundamental algorithms, and are used in all applications storing, processing and retrieving data. Therefore, it is not surprising that collections have been at the heart of Computer Science research since the inception of the discipline. From lists, stacks, queues through trees and maps to union/find data structures, a great number of *abstract data types (ADTs)* [90] and concrete data structure implementations along with efficient algorithms for the organisation and efficient retrieval of data have been developed over the years [35].

While individual data structures and algorithms for general data collections are well understood, there is less consensus about the relationship, or more specifically, the *hierarchy*, of different kinds of collections. The designers of collection abstractions for different programming languages have taken quite different approaches to organising collections in object-oriented class hierarchies. For example, the *C++ Standard Template Library (C++ STL)* [127], the *Java Collections Framework (JCF)* [106], and the *Scala Collections Framework (SCF)* [97] capture essentially the same collections in different ways. This is because existing class hierarchies and design patterns for collections are *operation centric*, where inheritance relationships dictate the structure. Furthermore, specific non-functional requirements, like the prescribed algorithmic complexity of certain operations in the C++ STL leave little choice when implementing STL containers. Despite superficial similarities among the collection hierarchies in the C++ STL, the JCF and the SCF, there are major differences between the frameworks and any programmer familiar with the fundamental concepts of one of the frameworks would need to spend significant effort to familiarise themselves with the other frameworks before becoming confident in their efficient use [27]. To illustrate this we refer to the JCF, where a *Stack* extends a *Vector*, which in turn implements a *List*; while a *LinkedList* implements both a *List* and a *Deque* interface. Figure 1.1 illustrates the hierarchies for both data collections, which appear non-intuitive in terms of semantic properties of the collections represented. This raises the question: why would a stack be a vector or a list?

The know-it before use-it process on interacting with data collections need to be avoided. If there is an abstract model that totally hide the implementational details of the data structure from the programmers, then the selection of the most appropriate data structure can be undertaken automatically. However, it is impractical to let the compiler fully decide the data collection, e.g., through type inference, without any information

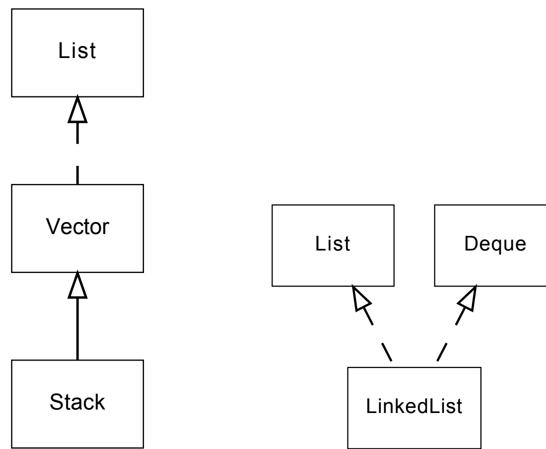


Figure 1.1: Excerpt from the Java Collections Framework, showing the inheritance hierarchy for a Java Stack (left) and LinkedList (right). Motivated by objected-orientated class inheritance where subclasses inherit methods from their parents, the resulting inheritance relationship fails to capture the *intuitive* collection properties. Typically, a programmer would not see a *stack* as a specialised *list*, but as a collection with the *LIFO* property, and *push* and *pop* methods for access.

from the programmer, at least for current technology it is not practical, or we can not make all the data structures a unified one such as all the data structures are lists. In fact, some minimal information from the programmer side is needed to help decide the concrete data structures. Thus, we identify and expose a set of properties of data structures, and model the abstractions for data collection based on the properties. We will discuss the properties and the property-based data collections in later chapters.

1.1 Solution: Programming with Collection Abstractions

The solution introduced in this section attempts to address the problems described earlier, which correspond to the three motivating examples to be described in Section 1.2.

In this thesis, we develop a novel approach to providing user-facing abstractions for data collections for writing parallel programs in an easier way. In our *Collection Skeletons* framework, programmers do not instantiate a collection of a certain class (e.g., `std::list<int>` for a list of integers), but instead, they explicitly specify properties that the collection must provide. The **key idea** is to entirely decouple specification from implementation of collections, thus avoiding the potential overspecification issues

from the non-specialised programmers' side. Instead of abstract or concrete data types for their data collections, programmers *only* specify the properties their collections are relying on, thus giving the Collection Skeletons framework the flexibility to select *any* concrete implementation that provides the requested properties, which also enables the portability to different platforms. For this, we distinguish between two kinds of properties concerned with 1. the *semantics* of collections, i.e., the expected behaviour, and 2. their *interfaces*, i.e., the available methods to access functionality. Although there are properties that do not belong to either semantics or interfaces, e.g., memory efficiency of a data collection, those *non-functional* properties are beyond the scope of this thesis and are subject of our future work. Unlike existing collection frameworks, Collection Skeletons do not attempt to fit different kinds of collections into a hierarchy, but instead we provide a single versatile and parameterisable collection type.

Semantic properties refer to the expected behaviour of collections. For example, whether or not a collection is allowed to store duplicate entries, or if it is restricted to unique data items, is a semantic property. *Interface properties* relate to the provision of specific functions to interact with the collection, e.g., whether or not a *split-by-value* operation for splitting collections around a pivot element is provided. We discuss these properties in detail in Chapter 4.

Shielding collections from implementation details enables us to transparently provide parallel collection implementations, which do not require any changes to the application's source code. This is because the parallelism is *implicit* to the collection implementation, whereas its external, declarative interface remains untouched. Another level of *explicit* parallelism is enabled by applying data-parallel algorithmic skeletons [32] such as *map* and *reduce* to collections. In Chapter 5 and Chapter 6, we show how both user-level parallelism and parallel algorithmic skeletons interact with Collection Skeletons, and characterise necessary and sufficient collection properties required for data-parallel algorithmic skeletons.

We have prototyped our novel Collection Skeletons framework as a C++ library (see Section 4.2). For its evaluation, we have rewritten several benchmark applications exercising collections of different kind and nature, and executed them on three different hardware platforms (an Intel NUC 10 desktop, a server with dual Intel Gold 6154 CPUs, and a 4-core Oracle Cloud Arm server, which will be introduced in detailed in Section 4.6). This enables us to measure any potential overheads introduced by our abstractions. In fact, we demonstrate that our Collection Skeletons introduce only negligible runtime overhead, and deliver performance improvements on several occasions.

This is because our framework enables us to select an implementation different from the collection used in the original code base, which results in higher performance. We also show that the best possible implementation choice for a given collection is program- and platform-dependent. This is where the flexibility of Collection Skeletons is of clear benefit: Concrete implementations can be flexibly swapped out without modification of the user's application code.

1.2 Motivating Examples

In this section, we present three examples, each of which highlights a problem with existing practice: non-specialised programmers find it difficult to write parallel code and they tend to overspecify the data structures, and how can it be potentially be solved by providing a higher level abstraction of data collections? We will start from a simple use case in C/C++ programming where the programmer uses a for loop to iterate over a collection of data, then we further discuss how to parallelize that piece of code and the challenges during the parallelisation. After that, we introduce how to rewrite a segment of code with algorithmic skeletons and the challenges in applying those skeletons. In fact, these simple everyday programming tasks inspired the abstraction for data collection proposed in this thesis.

1.2.1 Problem: Overspecifying / Suboptimal Choices on the Data Structure

This example illustrates that overspecifying the data structure can result in potential performance losses, even in sequential code. Consider the linked list example in Figure 1.2.

Listing 1.1 shows a code snippet with a loop traversing a user-defined linked list, which can be commonly found in C programs. The user specifies a concrete implementation, which is a singly linked and dynamically allocated list using a user-defined data type, i.e., a singly linked list where the element type is integer and the next pointer points to the next struct. For this problem domain, the programmer wants to traverse every element of the singly-linked list, and adds 1 to each element of the list. Self-defining a linked list through a struct is common in C and even C++ as that gives implementation flexibility and possible specific optimisation to the programmer. The traversal loop performs pointer-chasing to move from current element to the next

```c
typedef struct
    nodes{
  nodes *next;
  int data;
};

for(; p; p=p->next
    ) {
  p->data += 1;
}
```

Listing 1.1: User-defined list in C.

```cpp
std::list<int>
    nodes;

for(auto& i :
    nodes) {
  i += 1;
}
```

Listing 1.2: List using the C++ STL.

```cpp
Collection<int,
    Seq, Dup,
    Variable,
    Ordered> c;

for(auto& i : c) {
    i += 1;
}
```

Listing 1.3: Collection Skeletons.

Figure 1.2: Motivating example showing the evolution of a linked list collection and its traversal expressed in C, C++ STL, and eventually using Collection Skeletons. While the C programmer employs a user-defined linked list data structure, the C++ STL offers a pre-defined list collection. In contrast, using Collection Skeletons a programmer requests a collection by explicitly specifying the properties they rely on for maximal implementation flexibility.

element. The choice of implementation *implicitly* defines semantic properties, e.g., the order in which elements are stored and the possibility for duplicate elements. The user-defined linked list can sometimes be error-prone and hard to follow, and hence hindering further optimisation from multiple sources. Manually operating the pointer to the next pointer pointed to might lead to potential memory leak, and the error can be hard to debug or profile.

Rewriting the piece of code with modern C++ can greatly improve the readability and avoid the raw pointer chasing thus preventing from potential memory issues. Using the C++ STL as shown in Listing 1.2, the user utilises a `std::list` collection data type from C++ STL, and the explicit pointer-chasing from Listing 1.1 is replaced by a range-based for loop. This notionally introduces a *list* ADT, where the concrete list implementation is hidden behind an operational list interface. The user relies on the properties provided by the *list* ADT as defined in the C++ STL. However, it seems that there is some difference between the list introduced here and that one self-implemented before. Here a doubly linked list has been introduced instead of a singly-linked one, which mismatches the semantics of the problem domain. In fact, we say that there is data structure overspecification in this piece of code. Although there might be little to no performance difference in this sample code as it has a rather simple computational task, the data structure overspecification can cause substantial performance variation when the program reaches a certain scale. Thus, transforming C code to modern C++ code is not a trivial task, especially transforming those with pointer chasing operations to the modern C++ equivalent. The transformation process can be error-prone and potentially leading to data structure overspecification even misspecification. From this example, we can conclude that transforming C code to modern C++ code requires the programmers to have much background knowledge on both languages, especially, they need to understand the new features and the data containers. We can generalise this to broader legacy code transformation —the programmer needs to have a thorough understanding of both languages and be especially carefully on the data containers' consistency to avoid overspecification.

To improve the original C code considering multiple aspects, is there a way to avoid the overspecification issue? To avoid the overspecification or misspecification of the data containers, we can think about dropping the details of the data container the programmer is going to use —is it possible to have a higher level abstraction over the data container where the programmer does not need to know the implementational details? Thus, we introduce a novel way of interacting with data containers, as shown in Listing 1.3. The

user makes explicit the properties (e.g., storage order, sequential accessible) they request for their collection rather than considering the concrete implementation of the container. These properties act as a layer of abstraction over the data structures. To break down the declaration of the data container process, we request an *Iterable* property since we subsequently iterate over the elements one by one, and we choose the *Size* of the collection to be non-fixed once initialisation, since the number of elements contained in the collection is not statically known according to the problem domain, while we may rely on a specific storage *order* and allow *duplicates*, because part of an algorithm not shown in the examples requires these properties. These properties are consistent with the problem domain. Thus, the library or the compiler has the opportunity to generate the *well-specified* data container.

We refer to the *Iterator* property as an *interface* property, since it is related to the provision of operational facility through an interface function to iterate one by one over the collection. Being *Iterable* has also indicated that the collection is *Ordered* at the storage level.

This motivating example shows Collection Skeletons provide a convenient abstraction to collection data types, which avoid tedious and non-intuitive hierarchies of collections, but instead equip the users to specify exactly the properties they need for their application. We show in our evaluation at the end of Chapter 4, Chapter 5 and Chapter 6, how this abstraction improves performance and incurs only negligible overhead.

1.2.2 Problem: Overspecifying the Control Structure

In this example, we demonstrate that overspecification of the control structure can obscure potential parallelism, thereby diminishing its performance. Writing parallel program is not trivial, instead, even writing parallel code for a simple task is not an easy task for non-specialised programmers. When developing a parallel program, the programmer not only needs to deal with the correct specification of the data container, but also takes care of low level details, which is another cumbersome task for non-specialised programmers. For example, pointer based data structures and pointer chasing operations often impedes writing a parallel program. Running a for loop in parallel is a common way to improve performance in programs that have loops where each iteration is independent and can be executed concurrently; however, when the iterations are not independent, e.g., data dependency exists between iterations, then it is difficult

to parallelize a for loop, either by manually rewriting or by a parallel framework such as OpenMP. Data dependency often exists in pointer chasing based for loops [56].

What is worse, when designing a parallel algorithm for a pointer chasing structures, many low level details such as cache hit, should be considered. If a programmer wants to port their parallelised version of a pointer chasing for loop to another architecture, they need to have a good knowledge of numerous hardware details such as if this architecture supports Simultaneous Multi-threaded parallelism (SMT), which further hinders the programming efficiency of the software porting. Furthermore, such porting tend to be error-prone, which will ultimately impact the usability & correctness of the program.

Let us consider another example that is slightly different from Section 1.2.1.

```
while(ptr = ptr->next){
    ptr->data = ptr->data + 1;
}
```

In fact, this segment of code is quite similar with that in Figure 1.2, while we changed the for loop to a while loop and explicitly write the self adding to make it clear. Ultimately, a compiler will generate the same result for both pieces of code. The question now arises: how does one formulate a parallel counterpart?

Firstly, let us consider if we can have a powerful compiler that generate parallelised code —as modern CPUs provide instruction-level parallelism where multiple instructions can be executed concurrently, for example AVX512 enables the concurrent execution of 8 double or 16 single precision floating point calculations. Therefore, can the compiler compile this piece of code, optimise it for an instruction level parallelised one? Unfortunately, we cannot —though it seems to be a simple one layer while loop on a singly linked list of integers, it is actually a recursive data structure (RDS) traversal loop, which cannot be parallelised through instruction-level parallelism (ILP) optimizations from a compiler, because of the fundamental serialization and variable latency of the loop-carried dependence through a pointer-chasing load.

Nevertheless, there are still ways to parallelize this piece of code —through Decoupled Software Pipelining (DSWP) [135]. In this example, DSWP decouples the for loop into two non-speculative threads, and both threads together contribute to the correct result of the program equalling to the original for loop. Below is a split version of the while loop.

```
while(ptr = ptr->next) {
produce(ptr);
```

```
}

while(ptr = consume()) {
ptr->val = ptr->val + 1;
 }
```

the split while loops follow the producer-consumer model, where the first while loop traverses the linked list to produce traversed nodes, the second while loop consumes the produced pointers to do the multiplication operations. Additionally, a queue has been maintained to store pointers where the producer enqueues the pointers and the consumer dequeues the pointers. If the queue is full, the producer will block waiting for a slot in the queue; If the queue is empty, the consumer will block waiting for upcoming pointers.

In this way, the data dependency has been properly solved, e.g., the traversal thread can continue even when the computation thread stalls, and thus the traversal thread can have the opportunity to buffer data for the computation thread's consumption, which allows the computation thread to be relatively independent of the stalls in the traversal thread. This split rewriting decouples the behaviour of the computation and traversing of the pointers, making it possible for the compiler to generate code that leverage the instruction level parallelism.

However, DSWP does not solve the problem proposed earlier —with DSWP, non-specialised programmers still cannot write parallel programs. It is difficult for non-specialised programmers to manually split a piece of code into two pipelined producer-consumer model. And there is yet to be a compiler that automatically split a piece of code with DSWP. In fact, after being introduced for nearly 20 years, DSWP has not been applied to common compilers to implement automatic parallelism or by applications to perform source code rewriting.

There are many similar techniques like DSWP that help transform a piece of sequential program towards a parallelised one, however, just like DSWP, none of them solves the problem —the non-specialised programmers still find it difficult to write parallel programs, at the same time. Transforming sequential code to a parallel counterpart with a powerful compiler seems to be attractive. However, such compiler might never be developed. As mentioned before, compilers are not smart —they cannot optimise the program based on the information that they cannot access during compile time.

To help the programmers, especially the non-specialised programmers write parallel

code, it will be wiser to think from the programmer's side. For this example, let us consider if we can help the non-specialised programmers write split programs at the very early stage instead of writing the first pointer chasing for loop on a self-defined linked list: then there will be no need for a powerful DSWP-enabled compiler!

Let us go back to the first example in Section 1.2.1, as we have already got the higher-level abstractions for data collections, which means we helped the non-specialised programmers write improved code, is there anyway to make it perform parallel computation? The programmers specify the abstraction for the data collections they want, which leaves the compiler or the library some space of flexibility, as the concrete data structures are yet to be decided. It is possible, in this case, the specification will eventually lead to a concrete data structure that is easily parallelised or more suitable for a parallel for loop in this example. Then, no pointer chasing, no data dependency. In this case, with the application level optimised concurrent concrete data structures and thereafter enabled instruction level parallelism provided by the compiler optimisation and the modern CPU, the piece of code can harness the parallelism from the hardware and benefit from it.

1.2.3 Problem: Over-integration of Data and Control Abstraction

This example explains that current algorithmic skeleton implementations help with control abstraction but make the mistake of tying these to overspecified pre-packaged data types (e.g., SkePU's Vector, Matrix, and Tensor). With the previous two examples proposed a higher level abstractions for data collections, the programmers, especially the non-specialised programmers now can write better code without overspecifying the data structures as well as parallel program. However, the parallelism is limited to the concrete data structures and the hardware enable parallelism, such as instruction level parallelism provided by the platforms and the compilers. As we have already got the higher level abstractions for data collections, would higher level abstractions for algorithms cooperate with those and target for boosted performance?

Let us revisit Section 1.2.1 to determine if we can abstract the algorithm, such as the for loop, and optimise them for parallel performance. In each iteration, an element of the data collection is added by one; And there is no data dependency between concrete element as clarified by the problem domain and the analysis of the properties. Figure 1.3 describes the computational details of the piece of code.

From Figure 1.3, we can reorganise the program as 1. an *apply to all* method; 2. a

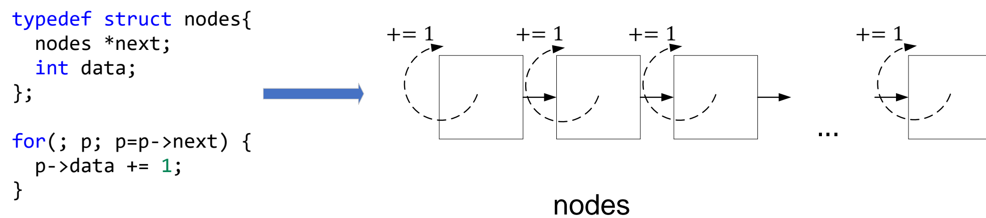


Figure 1.3: The computational details

function that accepts an integer as input and add that integer by 1 as a result. If we examine the algorithmic skeletons, it becomes evident that the operation is, in fact, a *map*.

As there are numerous implementations for algorithmic skeletons and parallel algorithmic skeletons, we replace the for loop with a map function of a portable implementation, and hopefully there will be performance boost if parallelised code is generated. Now, with the higher level abstractions of data collections integrated with algorithmic skeletons, non-specialised programmers can write more efficient code with less difficulty. We will discuss the integration of both abstractions in detail in Chapter 6.

1.3 Contributions

Overall, this thesis makes the following contributions:

1. In Chapter 4, we introduce a novel declarative approach to specifying data collections, exposing individual properties to be specified (rather than a pre-packaged sets of properties like in ADTs).

2. In Section 4.4, we identify a set of useful semantic and interface properties, which capture the key aspects of data collections programmers care about.

3. In Chapter 5 and Chapter 6, we demonstrate how parallelism can be exploited either through transparent concurrent data structures and parallel operations provided by Collection Skeletons, or be exposed through data-parallel algorithmic skeletons operating on Collection Skeletons characterised by suitable properties.

4. In Section 4.6, Section 5.4, and Section 6.5, we evaluate a prototype C++ library implementation of our Collection Skeletons framework against legacy bench-

marks rewritten to make use of our new abstractions, and demonstrate negligible performance overhead but good speed-up for both sequential and parallel configurations across three different hardware platforms.

1.4 Organisation of the Thesis

We have given an introduction as well as the motivating examples in Chapter 1. The rest of this thesis will be organised as six more chapters. Chapter 2 introduces background knowledge regarding this thesis, followed by Chapter 3 where related work will be presented. Chapter 4 introduces the Collection Skeletons, the property-based abstractions for data collections and the evaluation of the prototype library developed with C++ in a sequential case. This chapter is based on the published work from [52]. Based on the prototype library and the experimental benchmarks, Chapter 5 extends the library with Implicit Parallelism while Chapter 6 extends it with Explicit Parallelism, i.e., integration with algorithmic skeletons. Evaluation for both Implicit Parallelism and Explicit Parallelism will be presented in each chapter separately. Chapter 5 and Chapter 6 are based on our another published work [53]. In Chapter 7, we will conclude the thesis with research reflections and introduce future research opportunities based on this project.

Chapter 2

Background

This chapter provides the background knowledge for the proposed solution, encompassing data structures and their abstract model, i.e., ADTs. It introduces the hierarchy commonly observed in data structure libraries and delves into implementation strategies for parallel computing. After that, we further explore the algorithmic skeleton, which is a highly abstract programming model for realising parallel computing and has been extensively applied in various areas.

2.1 Data Structures and Abstract Data Types

This section introduces data structures and their higher level abstractions, i.e., ADTs. Data structures are an indispensable component of computer programming, and ADTs have become the de facto standard for implementing data structures or collection libraries in modern programming languages.

2.1.1 Introduction to Data Structures

In the field of computer science, data structures are of essential importance and are necessary in every program, regardless of the programming language used. A data structure is a format that organises and stores data in a computer [114]. It defines the rules by which the program accesses and processes the data enclosed. For example, a linkedlist defines the data to be linked one by one. Data structures are widely used in almost every sub area of computer science, from Operating Systems to the emerging ChatGPT [143], because the data structures are always organised properly, e.g., as a vector or a set. Data structures are used not only for storing or access, but also for

computational performance, storage performance or even energy-efficiency [74].

Data structures are of prominent importance from theoretical computer science to practical programming. However, choosing an *optimal* data structure is not always possible when developing a program where a series of algorithms are being applied to the data [140]. Data structures tend to be overspecified or even misspecified by the programmers, which can ultimately decrease the performance and even make the program more susceptible to errors. For example, a vector is appropriate for random access, whereas a linked list is appropriate for operations such as insertion and deletion. When a task involves intensive insertion over the data collection but a vector is applied, then it is obvious that the data structure has been overspecified. Consequently, the performance of loads of insertion operations over the vector will not be efficient thus resulting in worse performance. There are several ways to conquer the performance slow down caused by the data structures misspecified, e.g., by developing a novel and abstract data structure model to help *pick* the data structure more wisely, or by swapping out the misspecified/overspecified data structures during compile time or runtime. More abstract data structures have been proposed for many years and will be introduced in the following sections.

2.1.2 Abstract Data Types

ADTs provide a mathematical model for data types, which is defined by its semantics from the point of outside programmers [64]. ADTs serve as higher level of abstractions over data structures, with the concrete implementation details remaining transparent to the user. Although common programming languages, such as C/C++, do not support ADTs as they are formally defined, many data structure libraries of modern programming languages following the ADTs approach to implement and structure their data structures and their associated methods [18, 125]. With ADT, we can consider an abstract data structure, for example, an abstract linked-list, or a List, without defining the concrete details of the data structure on a real machine. To define an abstract List is straightforward: firstly, we should consider the behaviours of the List —what behaviours do we expect our List to have? We can note the functions that help alter the behaviours of a List,

1. append
2. head

3. ...

As we have written down the functions of the List without knowing the implementational details, under the concrete implementations, we can implement the List with a singly-linked list or an array as long as the provided functions are consistent with these functions.

2.2 Data Structure Libraries and Hierarchies

This section introduces common data structure libraries and their hierarchies of implementation. Many modern programming languages include implementations of ADTs as their data structure interfaces within standard libraries. For example, C++ offers the Standard Template Library, which encompasses implementations of common ADTs [127]; Java's Collection Framework provides a hierarchy of data collections as part of its standard library [106], and this is similarly exploited in other popular programming languages like Python and Scala [5, 97].

2.2.1 Scala Collection Library

Figure 2.1 presents the hierarchy of data collections excerpted from Scala 3.0's standard library, which resembles that of Java's data collections. These collections types are from the package `scala.collection`, and are all high-level abstract classes or traits, which generally have immutable and mutable implementations. There is already a multi-layer hierarchy from the top type *Iterable* to the bottom *BitSet*. As these collections are mainly for extension purposes for the concrete data collections implementations, the hierarchy presented here may not influence the programmability from the perspective of a programmer. As the difference of immutable implementations and mutable implementations is only the mutability of the data collection, i.e., whether a collection can be modified once initialised, we only introduce immutable data collections in this thesis while omitting data collections that only have a mutable version or an immutable version.

Figure 2.2 presents the hierarchy of the immutable data collections of Scala. From this figure, we can see that collection types have been primarily divided into three categories: 1. sequence 2. map 3. set. From the inheritance relationship among the hierarchy, the properties have been implicitly encoded into every collection, e.g., every data collection is iterable as each of them extends the trait Iterable. From the hierarchy

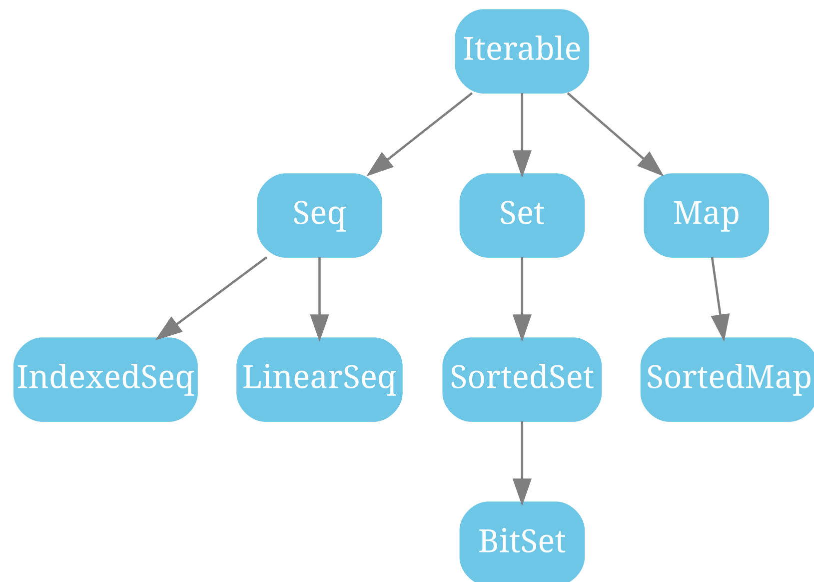


Figure 2.1: Hierarchy of Scala Collections [97]

presented in this figure, one can readily determine the properties of a specific data collection by tracing back through the hierarchy.

However, such a hierarchy is complex and primarily intended for library implementation purposes. When a programmer considers which data collection to select from the hierarchy, they do not lookup the hierarchy and the extension relationship, i.e., the implicit properties; Instead, they only consider the leaf node of the hierarchy and still need to explore the concrete implementational details of those data collections. In conclusion, the hierarchy of data collections, despite its implicit properties, does not provide programmers with higher-level abstractions for interacting with the data collections.

2.2.2 C++ Standard Template Library

The C++ Standard Template Library (STL) provides a collection of templates and generic algorithms, including common data structures and functions for C++ programming. Containers with the iterators, algorithms and other utilities are the core components of STL. Developed based on C++ templates, C++ STL provides compile-time polymorphism that is often more computationally efficient than runtime polymorphism implemented through inheritance [147]. For example, it is feasible to define a custom comparator and use it to sort a container with `std::sort`, and as the comparison logic can be defined at compile time, the sort operation should be efficient.

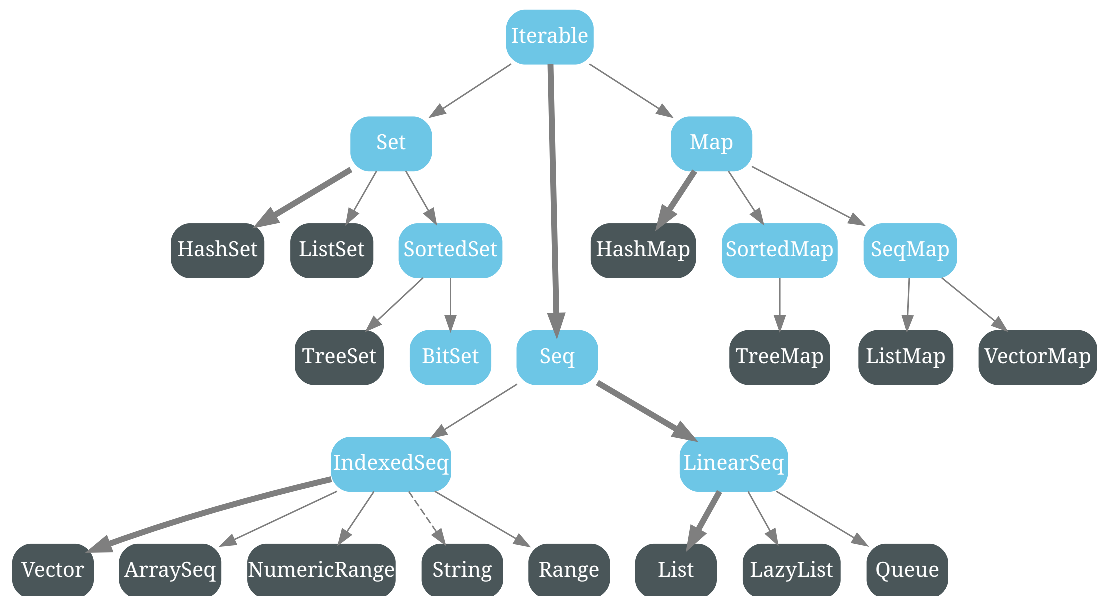


Figure 2.2: Hierarchy of Scala Collections —immutable collections [97]

Different from a hierarchy structure which is often used by inheritance, the organisation of STL's containers is flat —there is no extension or implementation relationship among containers. The STL implements ADTs and aligns with the C++ programming model. The STL offers fundamental containers such as vectors, lists, maps, and sets. These provide programmers with unified interfaces for handling data, e.g., eliminating the need to implement custom data structures or adapt to pointer-based arrays.

In modern C++ programming, utilising containers and algorithms from the STL is a standard practice. The STL benefits programmers in terms of reusability, efficiency, abstraction, and portability. There are several categories of containers in STL —sequential containers, associative containers, container adaptors and views.

- Sequential containers implement data structures which can be accessed sequentially.
- Associative containers implement data structures that can be quickly searched.
- Container adaptors provide a different interface for sequential containers to implement different data structures.
- Views offer flexible mechanisms for interacting with one-dimensional or multi-dimensional non-owning arrays of elements [15], a feature newly introduced in C++20.

Since containers are implemented using C++ templates, they can be wrapped or extended without compromising computational performance. Furthermore, the STL containers provide member functions to perform operations on data containers. The iterator-based element manipulation exposes the low-level implementation details to programmers. This allows them to operate directly on the elements without concerning themselves with pointer-chasing issues.

In STL, properties are also provided implicitly, as there are sequential accessed data containers and associative accessed data containers. Programmers have limited information regarding the properties of the data containers, and their ability to leverage these properties is also restricted. Besides, the STL also introduces time complexity for some operations over the containers [101]. The time complexity acts as a non-functional property to help decide a data container given specific operations over the data. However, despite these "non-functional" properties, programmers cannot readily leverage them when specifying a data container; they must still consider the implementation details before selecting a container.

2.2.3 Boost Library

Though powerful, the STL only provides a set of fundamental containers or algorithms. As a complement to the STL, the Boost library provides a larger collection of open-source, peer-reviewed C++ libraries that extend the functionality of C++ [87]. Several additional containers and algorithms have been provided in the Boost library. Moreover, most sub libraries in Boost are header only, thus programmers only need to include those necessary boost libraries without linking it thereafter.

In practical programming, whenever a programmer cannot find a data container or an algorithm in the STL, the Boost library is often the first alternative to consider. Following the similar development paradigm as of C++ STL, the Boost library is also portable across numerous platforms. Generally, the Boost library follows the same programming interface with C++ STL, making it an excellent complement in C++ programming.

2.3 Parallel Computing

As parallel computing has gained popularity across an increasing number of areas, there have been numerous parallel computing implementations, ranging from low-level

hardware to high-level applications [102]. Besides, there are also several standards to categorize the implementations of parallel computing, e.g., parallel computing on CPUs versus that on GPUs, or parallel computing for general purpose versus that for specific domains, etc.

From the perspective of programmers' practical development, parallel computing can be described as two forms, 1. Implicit parallelism, where the parallelism is transparent to the programmers, which require no programming efforts from the programmers; 2. Explicit parallelism, the programmers actively specify the parallelisation for their tasks. In practical parallel computing, several technologies incorporate both implicit and explicit parallelism. To describe these parallel technologies in the following sections, we introduce a spectrum where implicit parallelism is positioned on the left side, progressing toward explicit parallelism on the right side. Figure 2.3 presents this spectrum, showing the relative positions of various parallel programming frameworks that we will introduce in the following sections.

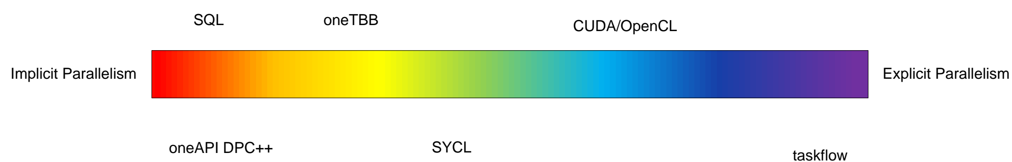


Figure 2.3: A spectrum for implicit parallelism and explicit parallelism

For modern architectures and parallel computing implementations, implicit parallelism and explicit parallelism can be provided at the same time. We will discuss this later in the following sections.

2.3.1 Implicit Parallelism

Implicit parallelism specifies a category of parallel computing where the concurrency and parallel execution is handled transparently, e.g, by the platform, the compiler, or a library, without requiring explicit coordination code or interactions from the programmer [138, 13, 8].

For implicit parallelism, the parallel implementational details with platform specific parallelism information are hidden from the programmers, thus the programmer can fully focus on the problem domain, i.e., the program logic of the code to develop for solving the problem domain. In the case of implicit parallelism provided by an automatic parallel compiler, programmers typically write the sequential version of the

program, and the compiler then generates code that runs in parallel for the specific platform. Implicit parallelism helps the programmer write parallel code. However, there can be substantial compromise on the performance.

Implicit parallelism tends to lack fine-grained control over parallel execution, as there is limited information that compiler or a library can retrieve. When runtime information cannot be fully accessed during code generation, the performance of automatically parallelized code from implicit parallelism may even be worse than its sequential counterpart. Furthermore, implicit parallel does not comply well with distributed computing environments as the communication can be too complex to handle in transparent to the programmers, and cause the program to be difficult to scale.

There are many implementations of implicit parallelism, mainly they are,

- Hardware accelerators such as GPUs [111] and TPUs [69] when low level parallelism is hidden from the programmers.
- Programming languages with their automatic parallelizing compilers, e.g., SQL [38].
- Parallel Computing frameworks, e.g., CUDA and OpenCL (they also provide explicit parallelism at the same time) [50].
- Parallel Computing libraries, e.g. oneTBB [119], thrust [9].

We provide brief introductions to the implementations of implicit parallelism most relevant to this thesis in the subsequent sections.

2.3.1.1 Intel oneTBB

Intel® oneAPI Threading Building Blocks (oneTBB, or previously TBB) is a flexible parallel computing library that simplifies the work of writing parallel programs across accelerated architectures including CPUs and GPUs [119, 86]. As a runtime-based parallel programming model for C++ code that uses threads, oneTBB help the programmers harness the performance of multi-core processors with minimum learning curve and programming efforts —similar to STL, oneTBB is also provided as a template library. OneTBB offers several concurrent data containers, and programmers can utilise them just as they do with standard containers from the STL. Compared to the sequential data containers in the C++ STL, these concurrent data containers are thread-safe for most integrated operations. OneTBB provides programmers with the opportunity to execute parallel operations without delving into low-level details, such as locks and mutexes, to ensure the thread safety of the parallel programs.

OneTBB is not a parallel counterpart to the C++ STL. In fact, only a subset of STL containers and algorithms have parallel equivalents in oneTBB. Rather than serving as a parallelized version of the STL, oneTBB primarily offers convenient task-based parallelization, allowing programmers to write multi-threaded code with ease. OneTBB also provides parallel implementations for some common algorithmic skeletons such as `parallel_for` and `parallel_reduce`.

OneTBB also supports nested parallelism, e.g., nested for loops to be parallelised and load balancing on the task-level (threads), making it a convenient and powerful library for the programmers to develop parallel code.

2.3.1.2 Intel® oneAPI DPC++/C++ Compiler

Intel® oneAPI DPC++/C++ Compiler is a cross-architecture C++ compiler that aims to produce optimised code that harnessing the hardware specifications such as the ever-increasing core count and vector register width in Intel® Xeon® processors [24]. DPC++ works as an automatic parallelism compiler to provide implicit parallelism to the programmers where they continue to develop code, compile & run just as they work with other C++ compilers; DPC++ generates code that has been optimised through several parallelizing strategies without efforts from them.

The Intel® Compiler helps the programmers boost application performance through superior optimisations and Single Instruction Multiple Data (SIMD) vectorization, which is also a popular parallel technique for instance by leveraging the SIMD OpenMP pragma. Furthermore, the Intel compiler is integrated with other Intel tools including Intel® Performance Libraries, together helping the programmers write parallel code for optimised performance. The Intel Compiler has also implemented the most updated OpenMP standard, which is a directive parallel programming model to be discussed later.

DPC++ has also included support for SYCL [34], which is an open parallel computing standard that help programmers to develop programs executing on heterogeneous architectures including CPUs and GPUs, with C++. As a complete compiler for C++, DPC++ is also perfectly compatible with oneTBB. With DPC++ and Intel's most recent hardware, the C++ programmers can continue their C++ programming without bothering to learn new features of a parallel library nor do they need to know the low level details of a specific parallel hardware.

2.3.2 Explicit Parallelism

Different from implicit parallelism where the parallelisation implementation is transparent to the programmers, for explicit parallelism, the programmers actively interact with parallel libraries, tools, or hardware to realise parallel performance [84].

Explicit parallelism requires more engagement from the programmers, which incurs more learning curve; however, it also provides more freedom thus more opportunities for more optimised parallelisation performance compared to implicit techniques. For example, by writing a program following explicit parallelism, the programmer can control several runtime factors such as the partition grain or the communication between tasks, thus achieving better performance for the specific problem domain.

There are also numerous frameworks or libraries for explicit parallelism. Algorithmic skeletons, which are introduced in the next section, represent a widely used abstraction for explicit parallelism. Taskflow [70] is also an explicit parallel computing library that implements task-based parallelism and provide the programmers with direct interactions with tasks. De facto standards of parallel programming, including MPI [139] and CUDA, provide explicit parallelism through their respective communication and thread models. When developing parallel programs with these frameworks, programmers must handle the hardware-related implementation details themselves.

2.3.2.1 Taskflow

Taskflow is a task-based heterogeneous parallel computing framework that provides user-friendly APIs to the programmers [70]. With Taskflow, programmers utilise task graph-based patterns to implement their parallel computational needs. When constructing the task graph, the programmers rely on a DSL-like grammar provided by Taskflow, to design the parallel working flow of tasks. Taskflow manages the scheduling and threading in the backend once the task graph is submitted.

Apart from the convenient task-based programming API, Taskflow also provides fine-grain parallelism control to the programmer also with a convenient way —it exposes in-graph control flow which enables end-to-end parallel optimisation.

2.3.2.2 SYCL

As mentioned earlier in Section 2.3.1, SYCL is an open standard that defines abstractions to enable heterogeneous device programming with modern C++ [78]. SYCL aims to help programmers develop parallel portable programs with minimum efforts. It enables

different heterogeneous devices to be used in a single application, where one source code runs on multiple platforms without substantial adaptations.

Like other parallel computing libraries for C++, SYCL uses generic programming from C++ templates as well as generic lambda functions to enable specifying higher-level abstractions for parallel computing. As an adaptable open standard for heterogeneous computing, SYCL allows for code generation across multiple backends, such as CUDA and OpenCL [41], using a single source code. And it can be integrated with more future backends and hardware with mild effort.

Currently, there are several implementations of SYCL which are being gradually applied to increasing areas. Intel oneAPI is one of the most promising and powerful SYCL implementations as it provides a more complete implementation for SYCL and targeting for various GPUs from vendors including Intel, NVIDIA, and AMD.

Comparing to established heterogeneous parallel computing frameworks such as CUDA and OpenCL that often require knowledge of low level details of the hardware platforms, SYCL provides a high-level approach as a unified programming interface to develop parallel programs and thus hiding the hardware details from the programmers. As SYCL aims to provide comparable programming expressibility as CUDA and OpenCL, the API is not provided as a higher level of abstractions. There is some learning curve for the programmers as they need to have an understanding of the programming model of SYCL which is closely related to the hardware. However, this also exposes more opportunities for fine grain optimisation from the programmers' side.

2.4 Algorithmic Skeletons

In this section, we will introduce the abstractions for another vital part of a program, the algorithm, as a complement to data collections. Algorithmic skeletons provide abstractions for a set of functional operations, which has essentially inspired this work on abstractions for data collections. Furthermore, there exists integration opportunity for the two different abstractions that will contribute towards an even more convenient and powerful programming model.

2.4.1 Introduction to Algorithmic Skeletons

Algorithmic skeletons are a high-level abstract programming paradigm that encapsulates common parallel programming patterns [32]. Abstract parallel programming patterns

are usually provided as functions, which hide the low level implementational details from the programmers. With algorithmic skeletons, the programmer, especially the non-specialised programmers, can write parallel programs with less effort. In fact, the algorithmic skeletons free the non-specialised programmers from the programming tasks, thus they can focus on their specific problem domain. Furthermore, with the development of numerous cross-platform implementations of algorithmic skeletons, there are several algorithmic skeletons frameworks that support heterogeneous platforms including GPUs, which means the non-specialised programmers can write GPU parallel program with algorithmic skeletons without needing to know any hardware details of those target platforms. Through implementations across various architectures, algorithmic skeletons offer significant portability in parallel computing.

Besides, depending on the programming languages and the implementations of the algorithmic skeletons, the pre-defined skeleton functions also expose further opportunity for performance optimisation. Meanwhile, the concrete implementations of algorithmic skeletons are inherently flexible, as they serve as high-level abstractions over common parallel patterns. This inherent flexibility facilitates easy scaling in distributed and parallel environments. The algorithmic skeletons have been widely deployed in many areas including scientific computing, deep learning, which will be discussed in detail in Section 2.5.

However, the algorithmic skeletons do not solve all the problems regarding writing parallel programs. Firstly, there exists a notable learning curve for non-specialised programmers; in relation to functional programming, these programmers must adapt to a functional mindset to effectively utilise algorithmic skeletons. This approach significantly differs from conventional programming methods. Additionally, while algorithmic skeletons encapsulate numerous common patterns in parallel computing, they do not address every problem regarding parallel programming. Programmers may discover that, after exploring all available algorithmic skeletons, they have to revert to traditional methods, which can compound the complexity of their programming task. Finally, if not implemented or utilized correctly, algorithmic skeletons can introduce significant overhead. For instance, in numerous implementations of algorithmic skeletons, invoking composed skeletons can lead to redundant I/O operations, thereby hampering the overall performance.

In the subsequent subsections, we will delve into three categories of algorithmic skeletons and explore their various implementations that are extensively applied in parallel computing.

2.4.2 Simple Data-centric Algorithmic Skeletons

Among various algorithmic skeletons, data-centric algorithmic skeletons are the most commonly used. These skeletons handle operations applied directly to the associated data collections. Consequently, they can be designed to be free of side effects, a characteristic preferred by numerous functional programming languages.

2.4.2.1 Map

A `map` skeleton applies a function to each element of a data collection independently. The example mentioned early in Section 1.2.1 is actually a `map` skeleton over the property-declared data collection. A `map` skeleton accesses elements within the data collection either in an iterative manner or in parallel, thereby enabling the function to operate on the accessed items. Figure 2.4 describes a `map` skeleton over a collection with a function `f`.

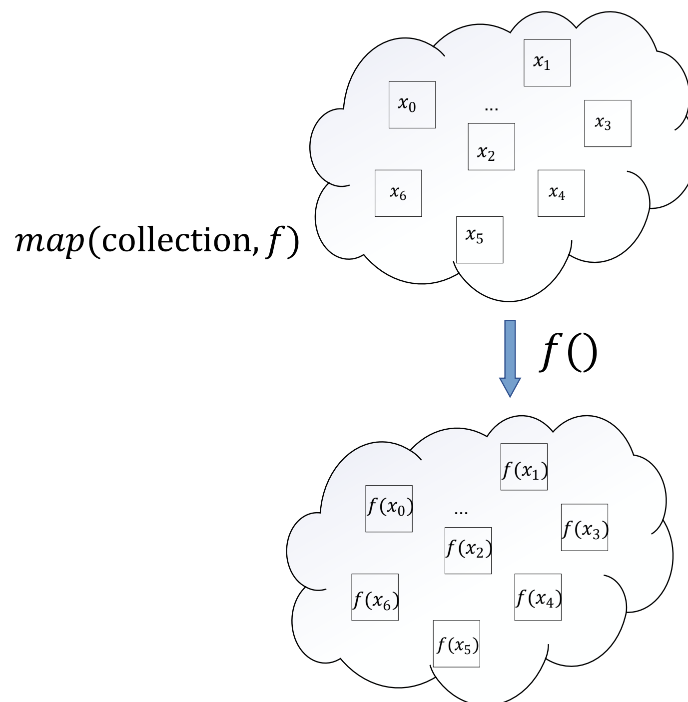


Figure 2.4: A map skeleton

2.4.2.2 Reduce

A `reduce` skeleton *sums* all the elements of a data collection into a single element using an associative function. It should be noted that the `sum` does not necessarily represent

an addition operator. Instead, it can denote any operation that aligns with the associative function. A **reduce** skeleton accesses elements within the data collection in an iterative manner. Furthermore, if the function taken by the **reduce** skeleton is commutative, then the `reduce` skeleton can also access elements in parallel to facility the function's operation. For example, Figure 2.5 presents a `reduce` skeleton over a data collection where the associative function is f.

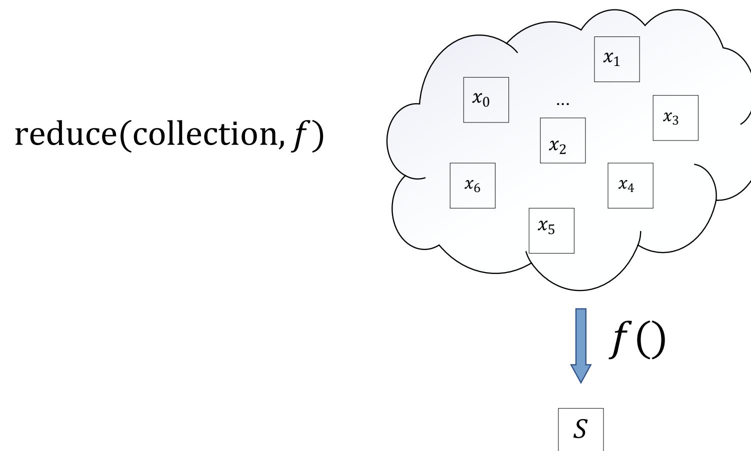


Figure 2.5: A reduce skeleton

2.4.2.3 Filter

Parallel algorithmic skeletons `map` and `reduce` are often applied together with `filter`, which applied a predicate function to each element from the data collection. If the predicate function returns true for an element, then that element is retained; otherwise, elements causing the function to return false are discarded. Similar to a `map` skeleton, a `filter` skeleton accesses elements within a data collection in an iterative manner or in parallel. Figure 2.6 describes a `filter` skeleton with a boolean function `f`.

2.4.2.4 Scan

A `scan` operates similarly to the `reduce`, but instead of returning a single element, it yields a data collection. Specifically, the `scan` conducts prefix computation by combining elements using an associative function. A `scan` skeleton access elements within a data collection in an iterative manner. However, as the definition suggests, a `scan` does not access elements in parallel (Although a parallel scan be implemented by

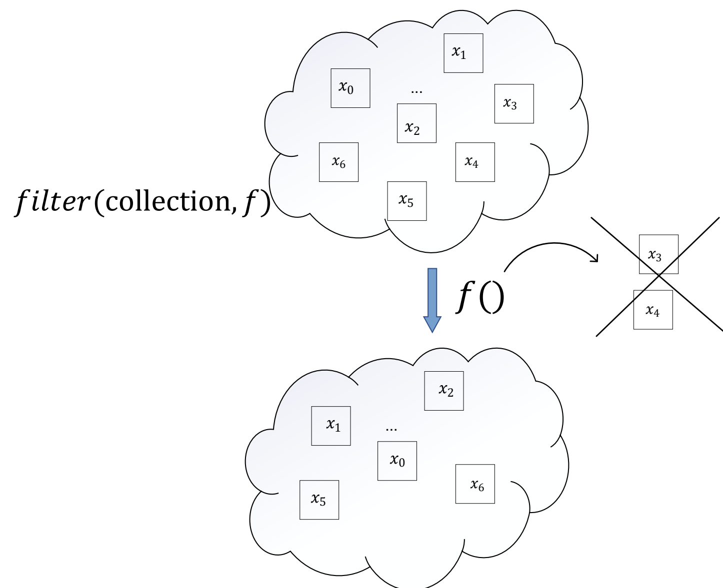


Figure 2.6: A filter skeleton

allowing copies of intermediate results). Figure 2.7 introduces a `scan` skeleton over a collection with a function `f`.

2.4.2.5 Zip

A `zip` combines two data collections element-wisely into a single data collection of pairs (or any kind of aggregates). Similar to a `map` skeleton, a `zip` skeleton accesses elements within a data collection in an iterative manner or in parallel. Figure 2.8 describes a `zip` skeleton "zipping" two collections with a function `f`.

2.4.3 Task-based Algorithmic Skeletons

Different from data-centric algorithmic skeletons that apply functions to the input data collections, task-based algorithmic skeletons focus on executing tasks in parallel. Task-based algorithmic skeletons are also widely used in many areas. However, their implementation tends to be more intricate than that of the previously mentioned data-centric algorithmic skeletons. Executing tasks concurrently necessitates synchronization among the tasks. Overheads, such as those resulting from communication, can adversely impact the overall performance. Moreover, synchronization among various tasks tends to be more error-prone compared to data-centric algorithmic skeletons.

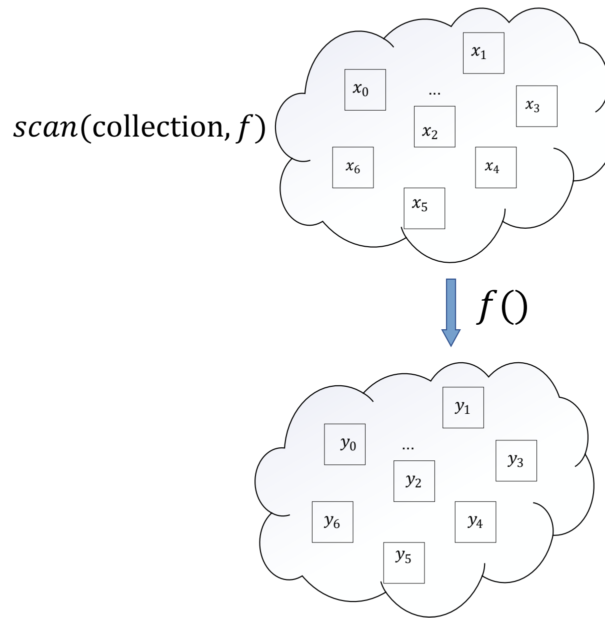


Figure 2.7: A scan skeleton

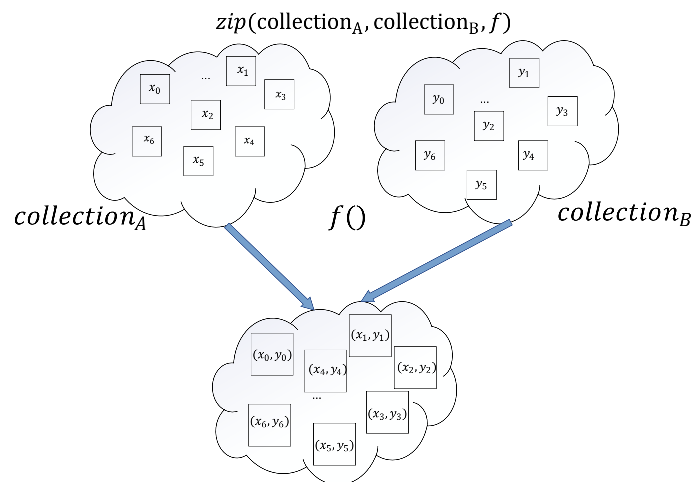


Figure 2.8: A zip skeleton

2.4.3.1 Divide-and-conquer

The foundational principle of the divide-and-conquer skeleton involves recursively subdividing a problem into smaller subproblems until they can be solved directly with efficiency [67]. Upon resolution of all these subproblems, their results are accumulated, ultimately producing the solution to the original issue. By processing these subproblems concurrently, one can achieve enhanced computational performance as opposed to addressing the comprehensive problem in a sequential manner. Figure 2.9 describes a `divide-and-conquer` skeleton over a problem `P`. By dividing `P` into several sub-problems and then solving these sub problems, the final solution `R` will be reached.

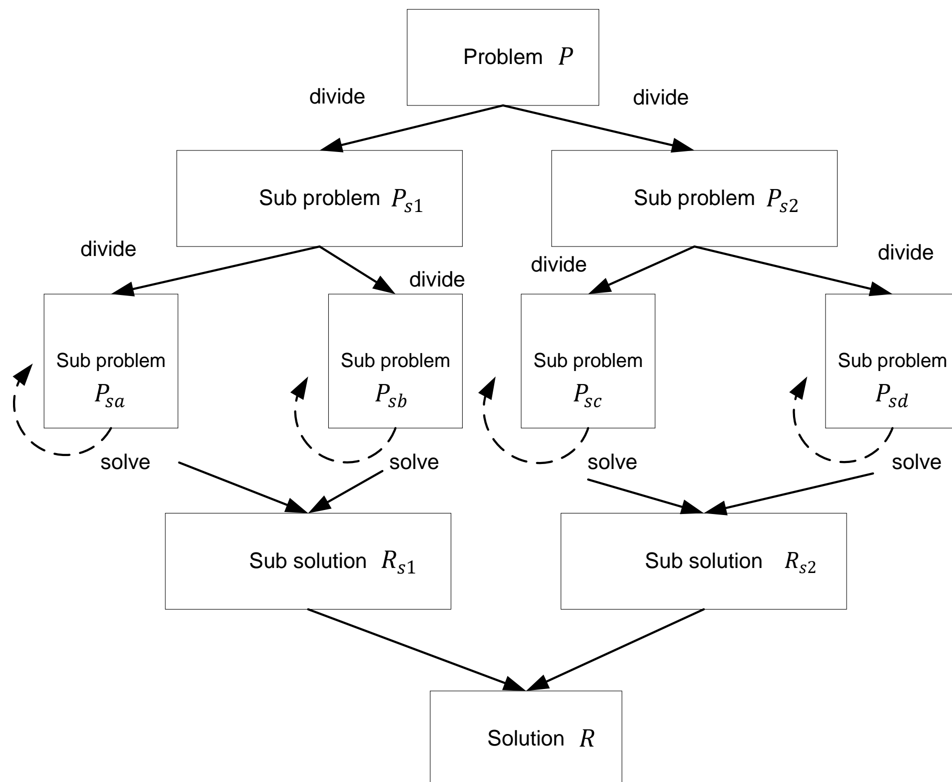


Figure 2.9: A divide-and-conquer skeleton

2.4.3.2 Pipeline

The *Pipeline* algorithmic skeleton organises computation into a sequence of stages, wherein data flows between successive stages in a manner analogous to a pipeline. The DSWP, as early discussed in the motivating example in Section 1.2.2, utilises this pipeline skeleton to realise parallelism.

Pipeline algorithmic skeletons function akin to stream processing. In this paradigm, the problem is partitioned into a linear sequence of stages. Each stage processes items from the input stream provided by the preceding stage and subsequently produces outputs for the subsequent stage. Contrary to the data dependency encountered in our motivating example, stages within a pipeline can be processed concurrently. For instance, utilising a shared queue can facilitate producer-consumer parallelism, allowing stages to run on separate CPU cores or GPU threads. Typically, stages in a pipeline are stateless. The pipeline algorithmic skeleton is extensively employed in various applications, notably in video processing, where videos are input as streams and segmented into different stages. Heterogeneous parallel frameworks, such as oneAPI and CUDA, incorporate the pipeline skeleton, ensuring its portability across heterogeneous architectures. Figure 2.10 presents a `pipeline` skeleton involving several process units.

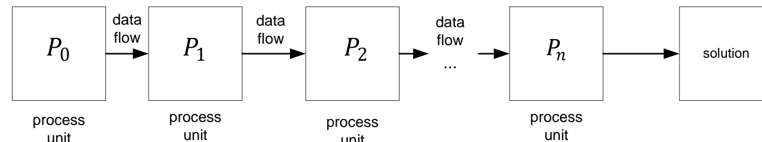


Figure 2.10: A pipeline skeleton

2.4.3.3 Farm

As its name suggests, a `farm` algorithmic skeleton has a master process that dispatches tasks to the worker processes to be executed in parallel. Unlike the divide-and-conquer algorithmic skeleton, the `farm` skeleton allows the master process to dynamically adjust the allocation of worker processes and manage load balancing among them. Worker processes are typically spawned and executed on homogeneous platforms; however, they can also span heterogeneous platforms, such as simultaneously utilizing both CPU and GPU. Analogous to the divide-and-conquer and other task-based algorithmic skeletons, the `farm` skeleton has been extensively adopted in various contexts and

has been implemented in several parallel computing frameworks, including OpenMP. Figure 2.11 presents a `farm` skeleton where a farmer delegates computational tasks to the workers on demand and those workers process the tasks.

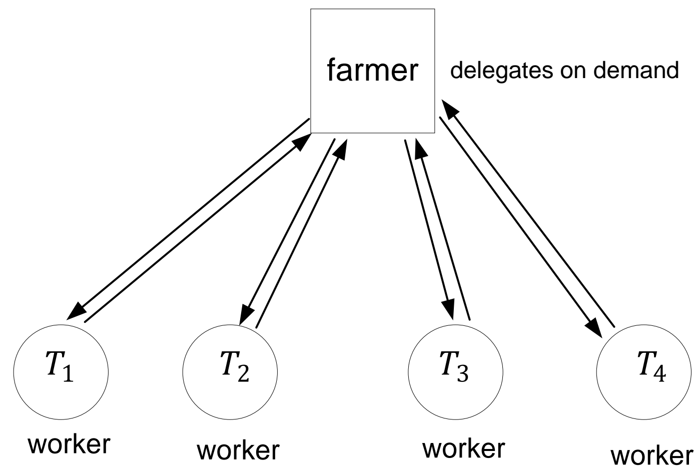


Figure 2.11: A farm skeleton

2.4.4 Other Data-centric Algorithmic Skeletons

There are numerous other algorithmic skeletons that have not been covered in this thesis. Beyond the basic data-centric and task-based algorithmic skeletons previously discussed, we categorize an additional set of data-centric algorithmic skeletons under the label *other*. These will be briefly introduced in the subsequent sections.

2.4.4.1 Stencil

The *Stencil* algorithmic skeleton is a widely-adopted paradigm, particularly prevalent in the domains of scientific computing and image & video processing [128, 43, 60]. A *Stencil* skeleton entails the concurrent update of each element based on a function of its own value and that of its neighbouring elements. This operation follows a regular pattern known as the stencil shape, which delineates the range of the neighbourhood. Additionally, during a stencil computation, coefficients are applied to interact with neighbouring elements. There's also a border policy in place that determines the update approach for elements situated on the data collection's border.

A stencil algorithmic skeleton usually operates on a multidimensional array or multidimensional vectors due to their grid-like shape. The stencil skeleton can also be generalised to graph and tree-like data collections. Like other algorithmic skeletons we described earlier, there are also numerous implementations of stencil algorithmic skeletons that perform on heterogeneous architectures, including CPU and GPU.

Some categories of stencil algorithmic skeletons can be interpreted as a map-like algorithmic skeleton, denoted as `Mapoverlap` [128]. A `Mapoverlap` skeleton specifies stencils where a function is applied to every element of the collection along with its surrounding neighbourhood. Figure 2.12 presents a common `stencil` skeleton where a computation on an element is decided by a function over its neighbour elements from four directions.

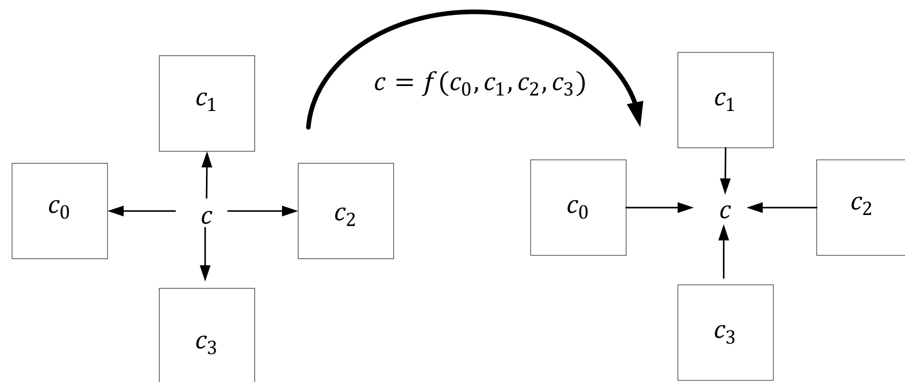


Figure 2.12: A stencil skeleton

2.4.4.2 Wavefront

The `wavefront` algorithmic skeleton operates by iterating over a data collection in a wavefront pattern, where a specific function is applied at each step [103, 104]. For a `wavefront` skeleton, the computations have a data dependency that forms a diagonal "wavefront" across the data collection. Usually a `wavefront` is applied to a two-dimensional grid, it can also be generalised to higher dimensions or other data structures such as a tree or a graph.

The `wavefront` algorithmic skeleton has been applied in various domains, particularly in numeric computing. Examples of its application include algorithms for pathfinding, dynamic programming, and Gaussian elimination, where the computation of an element often depends on data points from the predecessor iteration. Figure 2.13

presents a `wavefront` skeleton over a collection. The blue arrow is the computational direction of the wavefront.

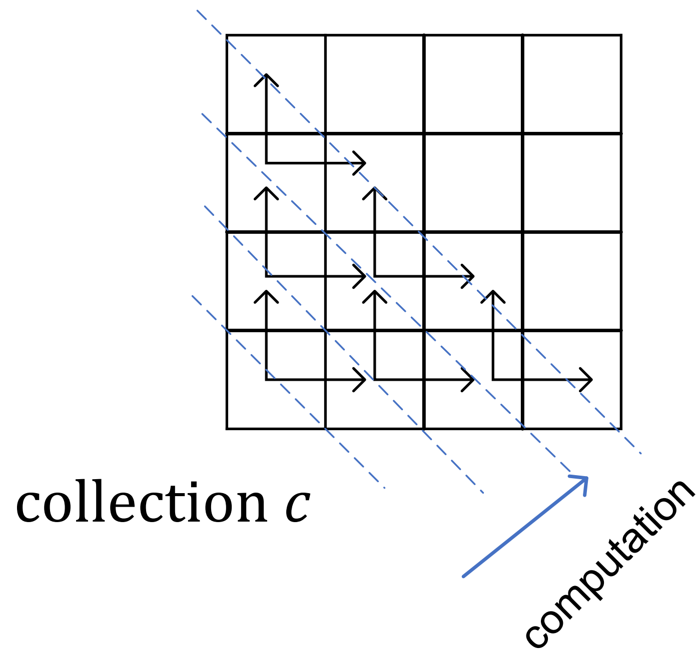


Figure 2.13: A wavefront skeleton

2.5 Skeletons-based Parallel Computing

Algorithmic skeletons are such convenient and efficient abstractions for heterogeneous computing that they have been widely applied recently in several areas regarding parallel computing. From the introduction to algorithmic skeletons by now, there have been developed several well-known implementations such as SkelCL [128], Muesli [31], SkePU [48], etc. Besides, modern programming languages such as Scala have provided a subset of implementations for algorithmic skeletons as part of their standard library. Furthermore, many parallel computing frameworks are not focused on algorithmic skeletons, e.g., Taskflow, have provided implementations for common algorithmic skeletons such as `map` and `reduce`, suggesting that algorithmic skeletons are vital in parallel computing.

This section will introduce some common implementations for algorithmic skeletons, of which some have inspired our work and some have been the backend of our prototype library to be discussed in the later chapters.

2.5.1 Algorithmic Skeletons in Standard Libraries of Modern Programming Languages

2.5.1.1 Algorithmic Skeletons in C++ STL

The C++ STL offers a comprehensive suite of algorithms, some of which share functionalities with algorithmic skeletons, albeit under different names. For instance, the `transform` algorithm in STL applies a function across a range of elements and stores the results in a designated range, mirroring the functionality of the `map` skeleton. The `transform` function can be parallelised in C++ by passing an execution policy as a parameter. Alternatively, one can utilize an automatic parallel compiler, such as Intel's DPC++, to compile the program. This compiler might generate parallel code, contingent upon the specifics of other parameters, such as the lambda function provided.

2.5.1.2 Algorithmic Skeletons in Other Programming Languages

Beyond C++, several other programming languages have incorporated specific algorithmic skeletons into their standard libraries. As mentioned early that Scala has provided a set of algorithmic skeletons as member functions within its collection standard library. The parallelism has been implemented through JVM's multithreading for those skeletons. Similarly, Java has also provided a set of algorithmic skeletons including the popular `map`, `reduce` and `filter`.

Dynamically-typed programming languages have also incorporated algorithmic skeletons into their standard libraries or as intrinsic features in recent years [58, 92]. Additionally, renowned Python libraries like PyTorch offer algorithmic skeletons that function across heterogeneous platforms [112]. This equips Python developers with the advantage of a dynamically convenient programming paradigm without sacrificing performance due to parallelisation.

2.5.1.3 OpenMP and MPI

The traditional and popular parallel computing frameworks including OpenMP and MPI also provide a part of algorithmic skeletons implementations. Open Multi-Processing (OpenMP) is a shared memory parallel computing framework that supports many platforms [42]. With the directive based programming interface, the programmers can easily parallelize the sequential code without much manual rewriting. In OpenMP, the programmer can specify a loop to be executed in parallel with a compiler directive, e.g.,

in C++,

```
#pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        a[i] = 2 * i;
    }
```

where the *parallel_for* can be viewed as an algorithmic skeleton [59]. There are other algorithmic skeletons implemented by OpenMP such as reduce and tasks operations, though with different names or signatures.

Messaging Passing Interface (MPI) is a standardised and portable message-passing system for parallel computing [139]. Different from OpenMP, MPI makes processes communicate with each other in a parallel computing environment, which is the standard for developing parallel programs in a distributed system. MPI standards have implicit introduced algorithmic skeletons, generally, the collective communication methods provided by MPI implement the functionalities of algorithmic skeletons with different signatures.

For instance, MPI's collective operations, specifically `MPI_Reduce` and `MPI_Scan`, offer standard MPI implementations of the `reduce` and `scan` skeletons, taking communication aspects into account.

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
   MPI_Datatype datatype,
                  MPI_Op op, int root, MPI_Comm comm)
```

where the collection to be applied with a `reduce` skeleton is implemented as a send buffer with a void pointer in MPI, while the result will also be stored into a receive buffer. It is worth noting that `MPI_Datatype`, `MPI_Op` and `MPI_Comm` are MPI objects.

Intel's oneTBB also implements algorithmic skeletons where they are provided as a set of generic parallel algorithms. The following presents a function signature of a parallel pipeline from oneTBB. It is evident that metaprogramming techniques, akin to those in the STL, have been employed. In fact, this thesis also leverages the parallel skeleton's implementations from oneTBB to implement the prototype library which is to be discussed in later chapters, while some minor modifications and wrappers have been applied to the original ones.

```
template<typename Range, typename Body>
        void parallel_reduce(const Range& range, Body& body);
```

In practical applications, OpenMP and MPI are frequently combined to develop high-performance programs. The algorithmic skeletons, regardless of their forms implemented in both standards, play pivotal roles in this context.

Apache Hadoop is a powerful, open-source framework designed for distributed storing and processing of large data sets [117]. Along with the widely used distributed file system HDFS, it provides a `MapReduce` programming model to process the data in parallel [45]. `MapReduce` is an extension based on algorithmic skeletons `map` and `reduce` and can be combined used with other algorithmic skeletons such as `filter`. `MapReduce` usually performs following a three-phase computation.

- Map phase, the library accepts input data and converts them into a set of intermediate key/value pairs in parallel.
- Shuffle/Sort, the library sorts the data according to the keys.
- Reduce, processes the intermediate key/value pairs from the Map phase and merges them, in parallel.

`MapReduce` offers exceptional scalability and fault tolerance, making it suitable for a variety of big data processing applications.

2.6 Chapter Summary

ADTs offer abstract representations for data containers, and there exists potential to delve deeper into abstractions that fully exploit the attributes of these containers. Algorithmic skeletons serve as parallel computing abstractions and have demonstrated their efficacy across numerous scenarios. Integrating algorithmic skeletons with property-based abstractions for data containers could enhance computational efficiency while preserving a clean programming model. In the subsequent chapter, we will delve into related works, placing emphasis on aiding everyday programmers in interfacing with data containers and crafting parallel code. A survey of existing solutions reveals a gap, underscoring the need for an enhanced approach.

Chapter 3

Related Work

This chapter introduces related work to this thesis. Related work can mainly be divided into five categories —1. Specifications or abstractions concerning data collections, focusing on the manner in which programmers interact with data, including the hierarchies of data collections. 2. Parallelization considerations in the context of pointer-based data structures, which is crucial for guiding programmers in crafting parallel code. 3. Techniques for detecting data structures in source code or binary forms. 4. Approaches to enhance a program via data structure substitution techniques. 5. Introduction to common implementations of algorithmic skeletons.

Each piece of related work has offered solutions within their distinctive problem domains and has significantly influenced this thesis. However, in addressing the challenge posed in Chapter 1, where especially non-specialised programmers struggle to develop parallel programs and face difficulties in specifying proper data collections, the existing literature falls short of presenting an effective solution.

3.1 Specification of Data Collections

There have been numerous efforts on the specification of data collections and higher level abstractions for data collections. [91] gives a survey on a number of specification techniques for describing data abstractions, as well as an evaluation on those techniques. In this paper, specification techniques including algebraic definitions, axiomic descriptions, arbitrary discipline, state machine model and fixed discipline have been discussed and compared. While these specification techniques do not explicitly address the properties of a data collection, they contribute to the property identification in this thesis.

[94] further introduces formal specification of an ADT `integer_list` in Fortran 90 and implements it with pointers based structures based on the specification. Their methodology of such formal specification can be generalised to other ADTs in Fortran 90 and even other programming languages including C/C++. Additionally, this paper concludes that achieving complete information hiding while retaining storage efficiency is not feasible when implementing the ADT using pointers in Fortran, highlighting the limitation between ADTs and pointer-based data structures.

In [11], a framework that uses Concern-Oriented Reuse (CORE) to capture different kinds of associations, their properties, behaviours, and implementation solutions within a reusable artifact has been proposed. This work focuses on software reuse, which also partly inspired our work and, in particular, aspects on portability. However, the properties are modelled within a hierarchy-based model, which is different to the techniques applied in this thesis. An extension of this work can be found in [123] where an abstract representation of a data collection for several possible solutions (concrete data structures) has been discussed for software reuse.

Algebraic data types are a fundamental concept and are prevalent in functional programming languages such as Haskell [81]. An algebraic data type is defined by combining other types thus it is a composite type. In functional programming languages, an algebraic data type is applied to create a data structure that store different data through a single unified type system. Based on algebraic data types, generalised algebraic data types (GADTs) allow more precise type annotations on constructors [28, 124]. GADTs can model complex data structures and their relationship accurately and ensure type safety for functional programming languages. Algebraic data types and GADTs focus on the correctness of operations of data types for functional programming languages rather than an abstraction for data structures, which is different from the idea introduced in this thesis. With algebraic data types and GADTs, the programmers still need to interact with the concrete details of the data structures and data types.

C++20 has introduced constraints and concepts to improve the expressiveness, safety and usability of metaprogramming [131]. With constraints, programmers can develop constrained templates where only types that satisfy the predefined concepts can be accepted. When designing a program where containers are involved, the programmer can explicitly specify the requirement that the containers need to meet through concepts on templates. Therefore, properties of data containers can be specified as concepts. However, the application of constraints and concepts are too complicated to programmers as metaprogramming with constraints is an advanced feature in modern C++.

To define a data collection with its properties through concepts, the programmer will inevitably implement many custom concepts as only a limited number of concepts are provided by the STL.

3.2 Collection Hierarchies

As mentioned in Chapter 1, modern programming languages such as Java [106] and Scala [97] implement a collection of data containers as part of their standard library following a hierarchy strategy —all containers are extended from the same top level data types and other types depending on the properties of them.The hierarchical organisation of data collections is strongly influenced by object-oriented design (OOD) and its use of inheritance, wherein common operations are placed at a higher level in the hierarchy so that data collections below can extend them. This operation-centric view of data collections sometimes gives rise to unintuitive aspects, particularly from the perspective of programmers. In the implementation of the collections, some properties have been utilised, e.g., *Iterable*. However, those properties remain transparent to the programmers. Programmers still need to be familiar with the details of the data container before interacting with them, which doesn't alleviate potential overspecification issues for them.

Other programming languages, such as C++, do not employ a hierarchical strategy for implementing the data containers in their standard library. However, a similar approach is still utilised. The C++ STL categorises containers based on their *properties* into three groups: Sequence Containers, Associative Containers, and Unordered Associative Containers. In addition, non-functional properties like access time complexity have been taken into account during the implementation of these functions. However, this approach doesn't entirely liberate programmers from the potential for overspecification, making it challenging to achieve parallel performance. Programmers must still understand significant details about the containers and their associated functions before using them. While the categorisation and time complexity considerations may benefit specialised programmers, they remain challenging for non-specialised programmers to grasp.

Designing a collection library based on ADTs for a programming language is non-trivial, given that ADTs lack a strictly formalized standard and programming languages have diverse implementation characteristics. In [96], collection libraries from 14 programming languages are reviewed. In their work, the authors discuss how to

design a collection library with multiple dimensions. The authors abstract collection types into five categories based on their semantics, including sequences, sets, maps, stacks and queues, and composed collections. They also discuss properties that are generally useful to for designing collection libraries, including,

- Basic properties, including the size of a collection, the insertion order and whether a collection sorts elements.
- Mutability, whether a collection is mutable or immutable.
- Multi-Threading Support, whether a collection support thread safety operations when performing in concurrent.
- Other properties, e.g., lazy collections vs strict collections, whether the elements of the collections are evaluated immediately upon creation, or they are evaluated until needed.

Additionally, [96] touches upon the co-design challenges between algorithms and data structures. It also delves into the utilization of collections, presenting preliminary statistics on the distribution of data collection usage. However, its primary audience seems to be library designers tasked with creating collection libraries for specific programming languages. Given that the properties of these collections aren't directly accessible to programmers, they can't leverage these characteristics for more efficient interaction with the library. Drawing inspiration from this research, our work aims to further illuminate the integration between algorithms and data structures. We advocate for a programmer-centric abstraction, which has culminated in the methodology presented in this work.

3.3 Parallelisation in the Presence of Pointer-Based Data Structures

Parallel programming has remained a challenging endeavour due to a myriad of factors, including compiler limitations, intricacies of programming languages, and more. Similarly, transitioning from a sequential to a parallel program presents its own set of challenges for the same reasons. Various frameworks and tools have emerged to aid in the conversion of sequential programs to their parallel versions. While many have showcased promising results within certain problem domains, the overarching issue

—that programmers struggle with crafting parallel programs —remains largely unaddressed. This section introduces various frameworks designed to parallelize programs with pointer-based data structures, where we pick one of the frameworks to discuss in detail. In exploring these techniques, which aim to convert pointer-intensive programs to parallelised versions, we deduce that the main obstacle is the average programmer's difficulty with parallel programming. Hence, a more programmer-centric approach is necessary.

3.3.1 Parallel Programming and Data Dependency

A data dependency is a situation where a program statement (instruction) refers to the data of a preceding statement. Data dependency is of vital importance in parallel programming. Automatic parallelisation is of efficiency only for parallelizing specific data dependency-free loops. Parallelising a program where a data dependency exists, often requires a substantial amount of manual work. Even worse, there are situations where manual work can do nothing to help parallelize a program because of the nature of the applied algorithm; for this situation, only a choice of a more parallelized algorithm can help. A number of profiling techniques have been proposed to discover the data dependency in a program to help parallelize it [39, 82, 83, 89, 85]. However, profiling techniques requires a profiler to investigate the program during its execution, which is more difficult to integrate with a common compiler in practice. Before profiling the program to perform data dependency analysis, pointer analysis must be applied to discover the data dependency. It is worthy to note that pointer analysis has a big problem as precise pointer analysis is an undecidable problem, which shows that a comprehensive pointer analysis on a pointer chasing situation is not possible [25]. In [85], a situation when pointer-chasing based access is presented, which ultimately causes the compiler not able to parallelize the inner for loop. By adapting a piece of code, i.e., changing the source code after discovering the data dependency, the compiler is able to parallelize the whole segment of program. However, for programmers especially those non-specialised programmers, it is difficult to discover the data dependence and alter the source code accordingly. [132] introduces evaluation strategies to separate the algorithm from the behaviour code in a program. Similar to algorithmic skeletons introduced in Section 2.4, evaluation strategies for parallel paradigms such as divide-and-conquer have been explored. Although evaluation strategies can be defined over all types in the language, the properties of the data structures are not considered within the interaction between

algorithms and strategies.

3.3.2 Decoupled Software Pipelining

[135] introduces the decoupled software pipelining (DSWP) technique, which transforms a for loop containing data dependencies into a structure more amenable to parallelization using existing techniques, such as task-based parallelism. DSWP partitions loops into long-running, fine-grained threads, organised in a pipeline using parallel computing models, such as the producer-consumer model. By decoupling inter-core communication queues using DSWP, certain pointer-based for loops can be parallelized, thereby achieving enhanced computational performance. Speculative techniques were introduced to DSWP as further enhancements in [135], allowing DSWP to be applied to a broader range of loops. The pipelining techniques employed by DSWP significantly restructure the code, making the impact of the transformation difficult to predict. Consequently, there are instances where the transformed code, despite being parallelized with DSWP, may exhibit poorer performance than the original sequential version.

Helix [22, 21, 20] was introduced as a solution to transform sequential programs into parallel versions, aiming to circumvent the shortcomings associated with DSWP. As a general-purpose automatic parallelisation strategy, Helix presents a novel code transformation mechanism and employs an effective feedback-directed heuristic to select loops for parallelisation. Helix-RC, an additional co-designed compiler has been proposed as an extension to Helix, leveraging specialized hardware and tighter compiler integration to further convert sequential programs into parallel equivalents [19]. However, both Helix and Helix-RC have limitations in their applicability as they can only parallelize a single loop within a sequential program.

In practical scenarios, programs often encompass more than just single loops and exhibit irregular patterns. To address the parallelization of such irregular programs, like those found in sparse linear algebra, Phloem [108] was introduced. This framework automatically identifies and leverages pipeline parallelism in these programs, eliminating the need for manual intervention by programmers. By decomposing the intricate transformation into a series of straightforward passes, informed by insights from manual applications, it produces efficient code that supports queue-based communication on Pipette architecture, which is a specialised architecture that enables cheap pipeline parallelism within each core [107]. While Phloem aims to offer extensive support for sequential programs, its reliance on specific hardware, as opposed to pure software

parallelization solutions, limits its portability, especially on heterogeneous hardware platforms.

Several works on data structure/pattern detection with the presence of pointer chasing based data structures have been proposed in [56, 57]. Utilising detection techniques to identify potential parallelism in a program can absolve the programmers from intricate parallelisation specifics, leading to enhanced performance when parallelism is discerned.

[57] introduced a technique grounded in pointer analysis and complemented by shape information to identify three parallel patterns: function-call, for-all, and for-each, specifically for recursive programs written in the C programming language. Such detection uncovers opportunities for parallelism in a program, offering advantages not only for programmers but also for compilers equipped with automatic parallelisation capabilities during code transformation.

[56] explored the application of pointer analysis to a wide range of areas within computer science, including scalar optimisations. [71] introduced a descriptive language designed to articulate the aliasing properties inherent in dynamic, pointer-based data structures. [72] put forward a general dependence testing method specifically tailored for dynamic pointer-based data structures. This method is capable of resolving false dependencies, such as those encountered within loop iterations, thereby paving the way for enhanced performance via code transformations, such as the conversion of a pointer-chasing loop. [63, 65] introduced a programming language mechanism alongside associated compiler techniques. These innovations allow the programmer to specify two distinct properties: speculative traversability and structural inductivity. This specialised programming language enhances the analysability of pointer-chasing data structures, especially those are outside scientific programs. While the properties introduced and the associated programming interactions offer limited advancements in programming efficiency, they serve as a foundational step for enhancing pointer-chasing based data structures through higher-level data collection abstractions. Before being extended to non-scientific applications, ADT description methodologies were employed to parallelize scientific programs that utilized pointers, as discussed in [73]. Several abstractions for recursive, pointer-based data structures have been put forth to identify potential optimization opportunities within imperative programs. Building on these abstractions, parallelisation strategies have been formulated and explored, including [64, 66, 62]. The Abstract Description of Data Structures (ADDS) facilitates the characterization of a data structure via its inherent properties, elevating the level

of abstraction. This, in turn, streamlines program analysis and fosters transformations aimed at enhancing performance, such as transforming a loop involving a pointer-chasing data structure into a software pipeline to realise parallelism. The study presented in [141] builds upon previous shape analysis and dependence analysis techniques, further extending their application to handle intricate recursive programs that are based on pointer-chasing data structures.

3.4 Data Structure Detection & Shape Analysis

Data structure detection is a useful technique that has been explored in various domains of computer science, including reverse engineering, computer security, and software maintenance. To replace a code segment in a program, such as a data container, it is essential to first identify its location and nature. The strategies for data structure detection can primarily be classified into three categories: 1. Memory profiling or memory analysis; 2. Shape analysis; and 3. Reverse engineering.

Data structure detection is also beneficial for parallel programming. By identifying the data structures within a program, programmers or automated parallel compilers can pinpoint locations for parallelization. Consequently, the detected data structures and their associated operations can be swapped out with their parallelized counterparts, as exemplified by the substitution of a recursive data structure [57].

In [79], the Data-structure Detection Tool (DDT) was introduced as a method to automatically identify data structures within an application. DDT operates under the assumption that memory accesses comprising a data structure are encapsulated by interface functions. Therefore, a data structure can be delineated by its memory graph, a set of interface functions, and the invariants associated with these functions. Experimental results indicate that DDT can accurately detect the majority of data structures, although there are exceptions in certain scenarios. Additionally, the time DDT spends on interface identification constitutes only a minor portion of the total overhead, facilitating its use in detailed data structure analysis during actual software development. However, a limitation of DDT is its strong reliance on the assumption that the time spent on interface identification constitutes only a minor portion of the total overhead.This limitation restricts its applicability in programs where compilers can apply inline optimization to functions manipulating data structures.

To address the limitations of DDT, Mempick [61] was introduced to detect and classify high-level data structures in stripped binaries. Mempick organises heap buffers

and their interconnections into a memory graph. It then segments this memory graph based on type analysis results and subsequently analyzes the shape of the data structures. Importantly, Mempick has the capability to differentiate among various balanced trees, depending on their usage within the corresponding application. Evaluation results suggest that Mempick can identify data structures with a high degree of accuracy.

Unlike memory-based analysis, shape analysis endeavours to determine the structure of a data structure by accessing it through a heap-directed pointer. [55] introduced a shape analysis method capable of discerning whether a data structure takes the form of a tree, a Directed Acyclic Graph (DAG), or a cyclic graph. While shape analysis offers simplicity and commendable accuracy, its applicability is constrained to a specific set of data structures, limiting its broader usage. Furthermore, while it is feasible to deduce a data structure based on its shape, shape information alone often proves insufficient. When a programmer needs to specify a data structure, additional information or properties are essential to comprehensively define a data collection.

3.5 Data Structure Replacement

Transparent replacement and dynamic switching of data collections have been discussed in several studies, such as [36, 80, 144, 37, 44, 140]. Commonly, these methods operate under the assumption that users have explicitly indicated their data collection preferences by selecting a specific container from a collection framework, even if the runtime behaviour remains unspecified. Subsequently, quantitative runtime data informs the selection of an alternative container. This alternative, while functionally equivalent, exhibits superior runtime behaviour for the given context. Notably, the majority of these studies focus on Java environments, where one data container can be directly substituted with another on JVM, where the JVM interprets or JIT the bytecode. However, in compiled programming languages such as C++, where the compiler compiles the source code into machine code, the use of various intermediate representations (IRs) by compilers from different vendors typically makes it difficult to perform such swapping as on JVM.

In [80], a novel program analysis tool has been introduced that selects the optimal concrete data structure for a program tailored to a specific microarchitecture. This tool applies a pretrained machine learning model to determine the optimal data structure, taking into account runtime characteristics such as application specifics and target architecture information, both of which are collected during runtime. Experimental

results indicate that this program analysis tool effectively optimizes the input programs while maintaining high precision.

Similarly, [144] acknowledges that the inappropriate use of containers can negatively affect a program's overall performance. In response, it introduces an application-level dynamic optimization technique named CoCo. This technique identifies the optimal container for a program and swaps it during runtime. However, the range of containers supported by CoCo as candidates is limited. Furthermore, the swapping rule is based on the container's size during runtime, implying that, in practice, it might be inefficient to replace certain containers in real-world applications.

In [37], an empirical study was conducted on the use of data containers across various popular Java projects. The study concluded that by merely altering the containers, there is significant potential for both computational and spatial performance enhancements in Java applications. Building on the findings from [37], [36] introduced an application-level framework designed to adapt data collections during runtime for optimized computational and spatial performance. This framework employs workload data at the collection allocation sites to inform its selection and replacement decisions. Moreover, an adaptive implementation has been introduced that can switch between concrete data structures based on the size of the data collection retrieved during runtime.

In [7], the concept of "Darwinian" data structures is introduced, referring to data structures that share a common interface but possess multiple implementations. The fundamental idea is that these data structures can modify their implementations while maintaining consistent interfaces. A framework named ARTEMIS was designed to identify the optimal implementation of a data structure and subsequently replace the existing one. Unlike the Collection Skeletons presented in this thesis, no abstract programming interface is provided for programmers in [7]. Instead, programmers must first define the concrete data structures and then allow ARTEMIS to optimise them.

Just-In-Time (JIT) data structures [44] are an attempt to shift the focus from finding the "right" data structure to finding the "right" sequence of data representations for optimised performance. This is achieved through a programming language to enable representation changes at runtime. However, this prototype language only supports a limited set of concrete data structures to be replaced by JIT, and there exists some overhead associated with determining the appropriate data structure and subsequently swapping it out.

In [140], the authors suggest replacing containers by analyzing a synthesized complexity metric of the original containers. This analysis is conducted using static tech-

niques derived from the source code, with the objective of selecting methods with reduced time complexity for each container under scrutiny. Contrasting with earlier literature, this approach operates at a finer granularity. It not only seeks to replace the container during runtime but also endeavours to pinpoint and substitute with the most optimal methods. Additionally, [140] highlights a collection of properties beneficial in determining the optimal methods through the static analysis.

3.6 Common Implementations of Algorithmic Skeletons

Programming languages that offer algorithmic skeletons typically implement only a selective set of these skeletons. Integrating a comprehensive set of algorithmic skeletons could render the language too cumbersome for general-purpose computing. For a more exhaustive collection of algorithmic skeletons, numerous frameworks and libraries are available, which can be integrated as third-party libraries.

3.6.0.1 SkePU

SkePU is an open-source algorithmic skeleton framework for heterogeneous platforms, including multi-core CPUs and multi-GPU systems [48]. To ensure compatibility with modern C++ programming, SkePU is provided as a C++ template library. Currently, it provides eight data-centric algorithmic skeletons that can be applied to four of the generic SkePU containers. It incorporates CUDA and OpenCL for GPU computing backends, ensuring compatibility with a wide range of GPUs from various vendors. Additionally, SkePU has recently expanded its scope by introducing a new backend based on MPI and StarPU [4], catering to distributed computing in cluster environments and thus broadening its applicability to diverse scenarios.

Below is a list of the eight algorithmic skeletons provided by SkePU. It is noteworthy that some of these algorithmic skeletons are compound, that are composed of multiple skeletons, which are particularly useful in specific application scenarios, such as stencil.

- Map: Apply a function to every element in the container.
- Reduce: Sum up the elements for 1D and 2D containers.
- MapReduce: An optimised version for map and reduce compound, as they are usually applied together.
- MapOverlap: Performs a category of stencil for containers from 1D to up to 4D.

- Scan: Performing generalised prefix sum to the container.
- MapPairs: Provides Cartesian product-style computation.
- MapPairsReduce: An optimised version for *MapParis* and *Reduce* compound, as these two algorithmic skeletons are also often used together.
- Call: Generic multi-variant component [48], which helps extend SkePU for computations that do not fit into any skeleton while still utilising SkePU's features such as smart containers and tuning.

Unlike the algorithmic skeletons in C++ STL, which operate on containers as long as they meet the requirements for iterators, SkePU offers four smart containers and restricts the application of its algorithmic skeletons to these provided containers: 1D vectors, 2D matrices, 3D tensors, and 4D tensors. These smart containers are shown in Table 3.1.

Table 3.1: Smart containers in SkePU and their descriptions

Smart containers	Description
Vector	1-dimensional vector
Matrix	2-dimensional matrix
Tensor3	3-dimensional tensors
Tensor4	4-dimensional tensors

When dealing with complex data input where those four smart containers can not handle, SkePU introduces proxy containers that help access multiple elements from a single smart container. Although the smart containers provided by SkePU abstract away many of the complexities associated with data management across heterogeneous memory architectures, the computational representation is limited to a small set of data containers. Furthermore, it remains imperative for programmers to be aware of the specific data containers they utilize when interfacing with SkePU.

3.6.0.2 eSkel

The Edinburgh Skeleton Library (eSkel) is a structured parallel programming library developed with C programming language as an implementation to the original algorithmic skeletons [10]. The eSkel offers a range of algorithmic skeletons that leverage the messaging passing parallel computing model, MPI. Though it is not actively developed

any more, there are some implementations and programming practice that are useful to this thesis.

In eSkel, there are no specific data containers tailored for the algorithmic skeletons. Instead, the programmers interact with data containers as they program with C programs, e.g., by self-defining pointer chasing data structures, or manually allocating a chunk of dynamic memory. The lack of versatile data containers in eSkel can negatively impact its user-friendliness and adaptability in various programming scenarios.

3.6.0.3 Muenster Skeleton Library

The Muenster Skeleton Library (Muesli) is yet another C++ implementation for algorithmic skeletons [31]. Similar to SkePU, eSkel also offers parallelism for heterogeneous clusters, encompassing multi-core CPUs, GPUs, and Xeon Phi coprocessors. Apart from data-centric algorithmic skeletons, Muesli also provides implementation for task-based algorithmic skeleton such as a task `farm`. It offers 1D array and 2D matrix containers that are easily portable to distributed clusters.

Similar to SkePU's smart containers, Muesli offers 1D array and 2D matrix containers that abstract from the memory hierarchy of heterogeneous clusters, facilitating coarse-grained parallelisation, and they are also presented as C++ template classes. These containers boast a flexible data (re)distribution mechanism, automatic memory management, and implicit (lazy) data transfer between different memory regions, thereby absolving programmers from delving into the details of the data container implementation, especially in a distributed setting. However, the application scope of Muesli's containers remains restricted, necessitating programmers to opt for either container type and necessitating a comprehensive understanding of these structures.

3.6.0.4 FastFlow

FastFlow is a C++ algorithm skeletons library for multi-core and distributed computing [2]. In addition to a set of high-level ready-to-use parallel skeletons, FastFlow also provides a set of building blocks to support low-latency and high-throughput data-flow streaming networks. With FastFlow, it is easier to write parallel programs as the programmers can model a parallel program as a structured directed graph of concurrent processing nodes.

While modelling the parallel processing has become more convenient, FastFlow does not provide containers associated with the algorithmic skeletons or building blocks.

The programmers need to work directly with the concrete data structures and carefully manage the interactions with the nodes within FastFlow.

3.7 Chapter Summary

In relation to the question presented in Chapter 1 —wherein non-specialised programmers struggle with the complexities of writing parallel programs and simultaneously face challenges with data container overspecification —none of the existing studies have offered a definitive solution. Building upon the foundational understanding and the literature reviewed, we present our proposed solution in the subsequent three chapters. This solution is designed to facilitate a more intuitive interaction for programmers with data containers, enabling them to more effectively write parallel code.

Chapter 4

Declarative Abstractions for Data Collections

This chapter commences with an introduction to our solution for the challenges delineated in Chapter 1 —namely, declarative abstractions for data collections. Subsequently, we delve into the properties that facilitate these abstractions. Following this, we explore the development of a prototype library for Collection Skeletons, utilizing C++ metaprogramming techniques. After evaluating our prototype library against 17 benchmark programs, we ascertain that it introduces minimal to no overhead; rather, it presents potential avenues for performance enhancement. It's noteworthy that the evaluations in this chapter focus solely on sequential implementations, deferring discussions on parallelisation to the subsequent two chapters.

4.1 Introduction

As early introduced in Chapter 1, to help the programmers better specify data containers thus facilitating Data-Centric Parallelism eventually, an abstract data collections library can be of great help. Furthermore, this data collections library must be portable, ensuring its functionality across diverse platforms. This is essential to effectively leverage the heterogeneous parallelism offered by various architectures, including multi-core CPUs and GPUs from multiple vendors. As an abstraction aimed at programmers, such a data collections library must also offer considerable flexibility, minimizing the programming effort required. For instance, programmers should not have to change their source code to adapt it to another platform, nor should they be obligated to extensively modify their existing code to utilise the library. Ensuring such high flexibility for programmers

implies that the library should handle the intricate details of implementation, shielding most of these complexities from the users. Crucially, the introduction of this data collections library should not lead to significant performance disparities. When other factors remain constant, a program coded traditionally should perform comparably to one written using the data collections library. This library must maintain performance for sequential programs, thereby offering robust opportunities for performance enhancement when porting the program to parallel architectures using the same abstractions. The library will benefit the non-specialised programmers as well as the broader users with intuitive abstractions and robust performance.

This thesis introduces **Collection Skeletons**, allowing programmers to declare a collection based solely on their desired properties for storing data. For instance, if a programmer wants a data collection with property *X*, they merely specify *X* as a requested property for the collection. The details of the implementation remain transparent to the programmer, eliminating the need for them to understand or manage at this stage. Interacting with data collections in this manner conceals potential parallelism from the programmers, while revealing it for internal optimization within the Collection Skeletons library. Indeed, substantial enhancements can be achieved concerning data-centric parallelism with the Collection Skeletons. This can be either through implicit parallelism, which remains entirely transparent to the programmers, or explicit parallelism, accessible to them via integration with Algorithmic Skeletons. Both types of parallelism will be explored in subsequent chapters.

Inspired and based on ADTs [90], Collection Skeletons compete with the concept of ADTs for the specification of data collections. As introduced in prior chapters, ADTs are mathematical models for data types, which specify user-facing signatures and semantics of operations on a data type. Specific properties of ADTs are expressed through the semantics of the operations it provides, e.g., for a *stack* we might expect a definition that captures that a *pop* operation following immediately after a *push x* returns the value *x*, and the state of the *stack* to be the same as it was before. In this sense, ADTs *implicitly* define properties through semantics of its operations. In addition, there is no notion of relationship between ADTs. For example, programmers might perceive *stacks* and *queues* as related collection data types that only differ in their data retrieval order, but their respective ADTs do not attempt to establish this similarity, thus outside users would have no information about the operations based on the names suggested by a specific ADT. Figure 4.1 presents a stack and a queue, where the inner storage are the same and both ADTs have similar methods to *push (enqueue)* and *pop*

(dequeue) elements. In fact, the only difference in this example is the in-out order, which is implicit in the ADT model, thus inaccessible to the programmers.

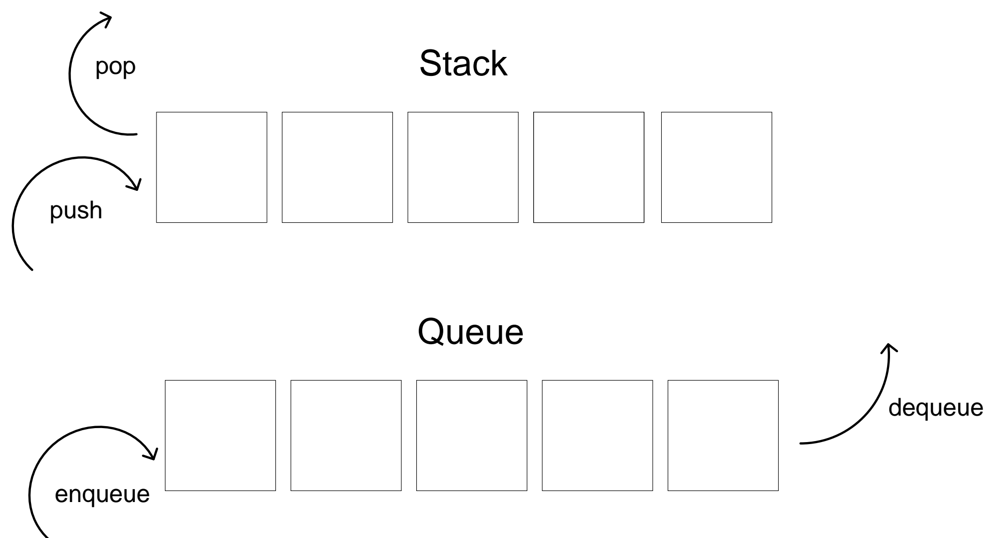


Figure 4.1: Stack and Queue as ADTs

Collection Skeletons, however, follow a different approach, where all collections are derived from a single, versatile *Collection* type by parameterisation. The parameters to this *Collection* archetype are semantic and interface properties, respectively. Semantic properties relate to e.g., whether data elements are unique or if duplicate entries are allowed, whereas interface properties specify available access functions for the programmer to interact with the collection. This mechanism of properties organisation enables the users to declare a data collection with the wanted properties, and, most differently from the ADTs, to retrieve information about a specific data collections declared with a combination of the requested properties. Through this way, the program can be more abstract at the implementational level but more clear at the source code level, which has the potential to benefit the programs in aspects of performance and software engineering.

4.2 Overview of Collection Skeletons

Based on the semantics from the ADTs and the interface functions that are widely used on standard libraries from popular programming languages including C++, Java and Scala, we propose eight preliminary groups of properties to help model the Collection

Skeletons in our prototype library. Properties belonging to the same group exhibit similar features. Table 4.1 presents the eight groups of properties, their definitions, and the API parameters which are abbreviations to the properties for convenient use in our library. By selecting properties from the table, programmers can tailor data collections to possess the exact attributes they desire.

Using these *declarative* abstractions for data collections, it is straightforward to request a data collection by specifying only the desired properties. We integrate the property-based model with modern C++ programming practice, making the learning curve for the programmers as smooth as possible.

Figure 4.2 presents the overview process of Collection Skeletons —as discussed earlier, the programmer requests a data collection similar to a database query, with the properties they desire the collection to have. The query is processed directly by the Collection Skeletons, thereby abstracting further details from the programmers. The Collection Skeletons then decide a concrete data structure that to be eventually generated in place of the property-based declaration. The technical details will be discussed in detail in the following of this chapter. Let us view the decision process as a black box. Fundamentally, this black box should yield one of two outcomes: either a specific data structure is determined, or a compilation error is triggered. The black box also has hidden opportunities for flexibility as there can be many candidates for the decided concrete data structure, which is still hidden from the programmers but requires the Collection Skeletons' developers to design, which will be discussed later shortly. Furthermore, the black box has hidden opportunities for parallel computing, e.g., concurrent or parallelised concrete data structures can also be the candidates. Such hidden parallelism is to be discussed in Chapter 5.

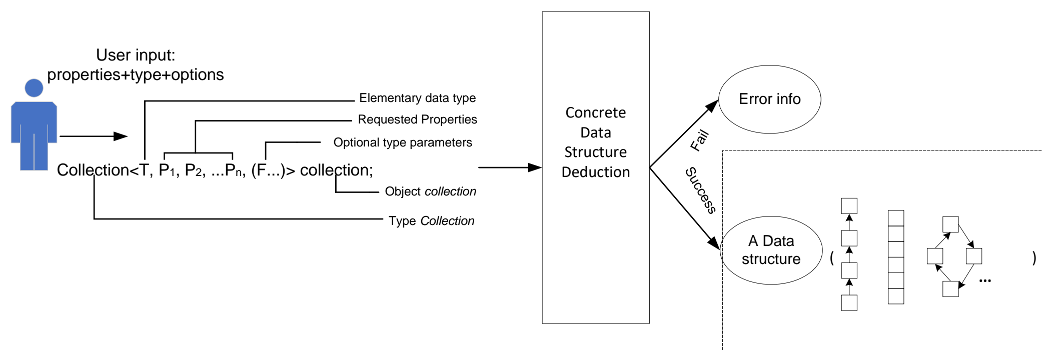


Figure 4.2: Overview of the Collection Skeletons' process

For illustration purpose, our prototype implementation employs `C++` template

metaprogramming, in which a collection can be defined as:

$$\texttt{Collection<T,}P_1\texttt{,}P_2\texttt{,...}P_n\texttt{,(F...)>}$$

where T is the elementary data type of the collection and P_1 to P_n are parameters from API column of Table 4.1. F... are optional parameters that might be specified by the user when necessary, which will be later discussed in detail. Using this approach, based on the properties specified by the programmer, the Collection Skeletons determine the most appropriate concrete data structure. This determination is made using an algorithm (for instance, through a pattern-matching technique based on the properties of concrete data structures, which we will discuss later in the context of the prototype library). Once decided, this data structure serves as the concrete container to store the data. Figure 4.3 illustrates the working process of a property-based declaration and the concrete data structure deduction. The collection type c1 is defined with properties as a type alias; when an object of c1 is initialised, the concrete data structure is deducted and returned.

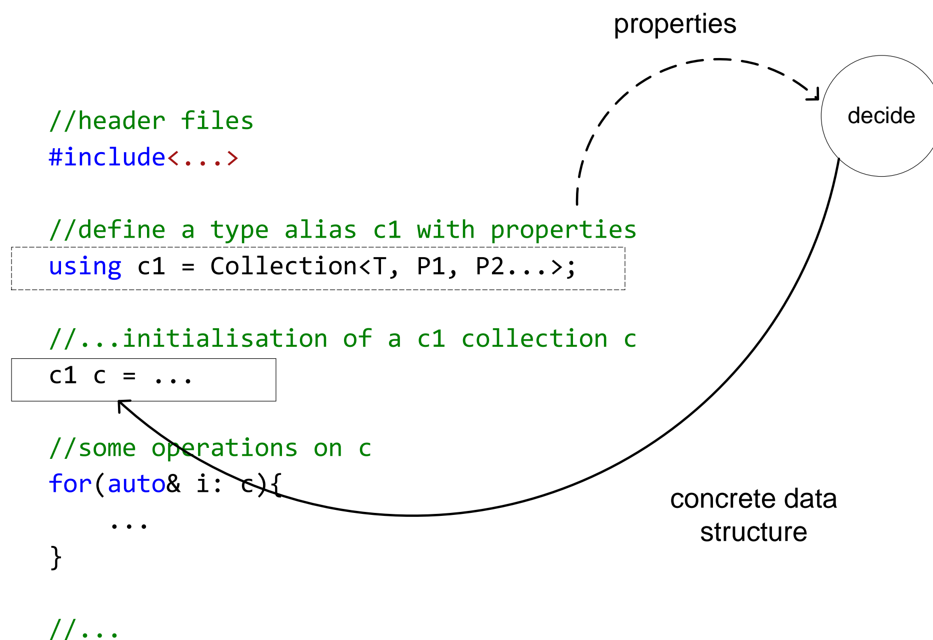


Figure 4.3: Example: properties and concrete data structure

Figure 4.4 describes the layers of an application where the Collection Skeletons are introduced. In this section, we outline the application stack, from the application source code layer down to the architecture layer. The application layer is where programmers describe their programs. With Collection Skeletons, they can define data collections using desired properties while keeping other parts of the program unchanged. They can

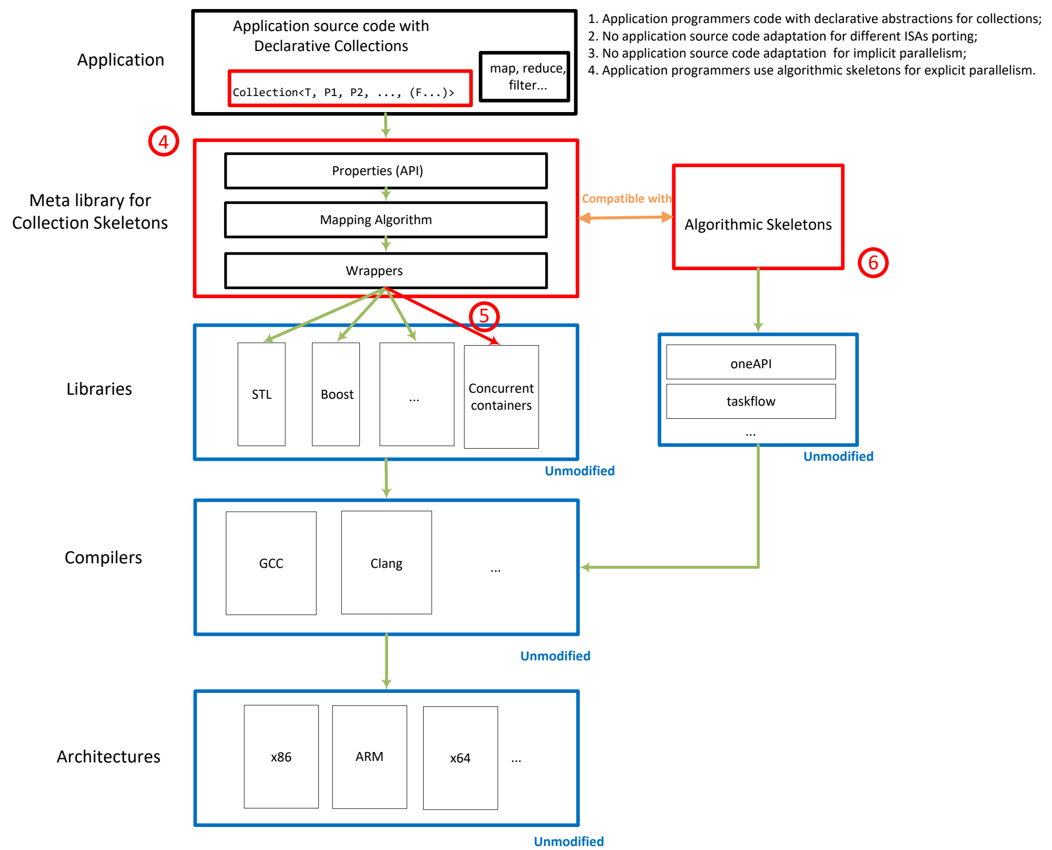


Figure 4.4: Overview of layers of an application with Collection Skeletons

also introduce algorithmic skeletons, which will be discussed in detail in Chapter 6. The meta library layer is where the Collection Skeletons reside. Rectangles with red borders indicate the contributions made by this thesis, while circled numbers refer to their corresponding chapters. Layers below the meta-library layer remain unmodified and are marked with blue-bordered rectangles.

This figure shows that the Collection Skeletons library functions as a meta library between the application layer and concrete data structure libraries. As a result, porting a program developed with Collection Skeletons requires no modifications to compilers or architectures, since concrete data structure deduction is implemented using C++ metaprogramming. This approach ensures a convenient programming model for developers while providing good portability and flexibility for further optimization.

To better understand how the prototype library for Collection Skeletons works, we will revisit the examples from Section 1.2 and demonstrate the advantages of using properties as abstractions for data collections.

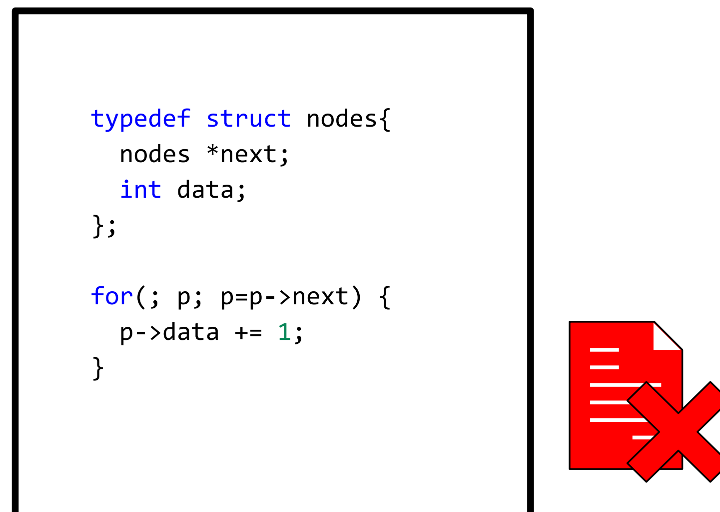


Figure 4.5: Not recommended: Self-defining a pointer chasing data structure

As discussed in Section 1.2, self-defining a pointer chasing data structure can be complicated and error-prone, as demonstrated in Figure 4.5. While the introduction of a container from the STL can be helpful, the application programmer still needs to understand the details of the data container, or they might take risk of overspecification or misspecification, as shown in Figure 4.6. Instead, with Collection Skeletons shown in

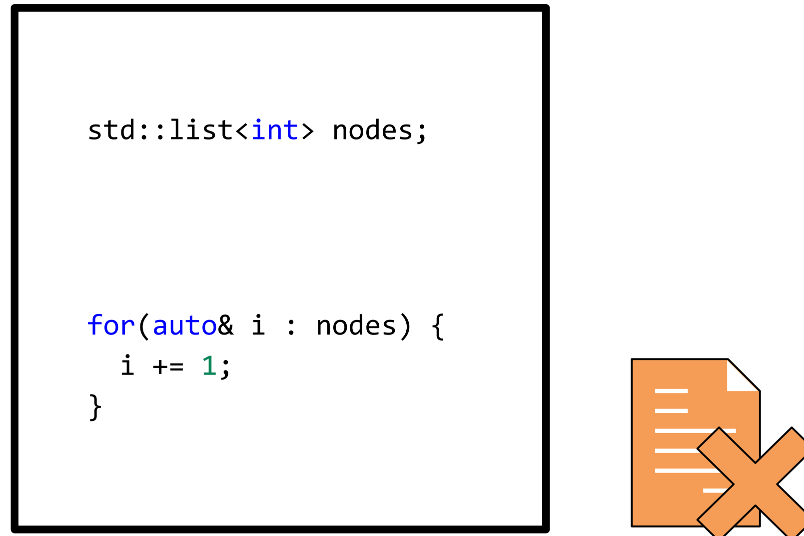


Figure 4.6: Fine, but can be improved: Using a list container from the C++ STL

Figure 4.7, the application programmer only needs to request the properties they want as template parameters in C++, and interact with the representation without considering the concrete details of the representation, because the library of Collection Skeletons will implicitly handle all the complicated details.

Figure 4.8 expands to Figure 4.7 with more details. With the prototype library, the properties declared in the API are expanded into concrete implementations with C++ template substitution according to a specific algorithm, which is to be introduced in Section 4.5.3. Meanwhile, a property compatibility check will be performed between algorithmic skeletons and Collection Skeletons, which is to be discussed in Chapter 6. By abstracting the declaration of data collections, great flexibility has been introduced as the library is able to transparently choose the concrete implementation based on different factors. For example, the library can choose different candidates for different architectures, or it can generate parallel data structures when applicable. We will explore this flexibility further in Section 4.5, Chapter 5, and Chapter 6.

4.3 Properties Identification

As mentioned in the previous section, ADTs inherently define properties through the semantics of their operations. Let us delve deeper into the properties implicitly defined

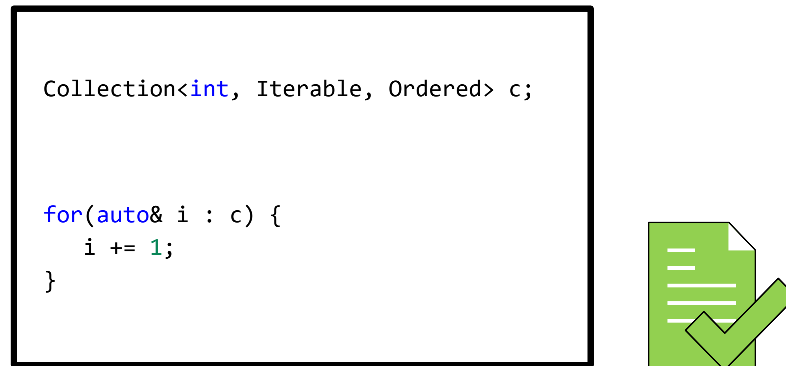


Figure 4.7: Excellent practice: Declaring data collections with the properties

by common ADTs and attempt to categorise them. Moreover, since data structures are much more than ADTs, we will also explore a variety of other data structures from diverse sources. Given that a data structure fundamentally organises and stores data on a computer for efficiency, we will scrutinize how data is organised, stored, and modified.

We will examine data structures from the STL containers, Boost library, Java collection library, and Scala collection library. While all these libraries implement ADTs, they differ in technical specifics. Therefore, we will first explore ADTs and subsequently delve into the aforementioned implementations.

Figure 4.9 presents the process of the property investigation. When looking into an ADT, e.g., a *List*, we first go over its definition, which is a *List* ADT representing a **finite** number of **ordered** values, where the **same value** may occur **more than once**. Based on the definition, we write down the critical keywords marked as bold and then look into its associated operations/interface functions that access or modify the elements. For example, there is a function `append` to add an element at the end of the *List*, then we write down *Appendable* and *Variable* as it can be appended through a function and thus its size is variable. Similar methods are applied to other ADTs and their associated operations. Figure 4.10 presents the ADTs that haven been studied and their properties discovered.

Following the similar methods, we continue to investigate STL containers, selective Boost containers, Java/Scala collections and other frequently used data structures. Since there are a great many data structures in these libraries, time constraints have forced us to focus our work on a subset of them.

By examining the identified properties of data structures, it becomes feasible to

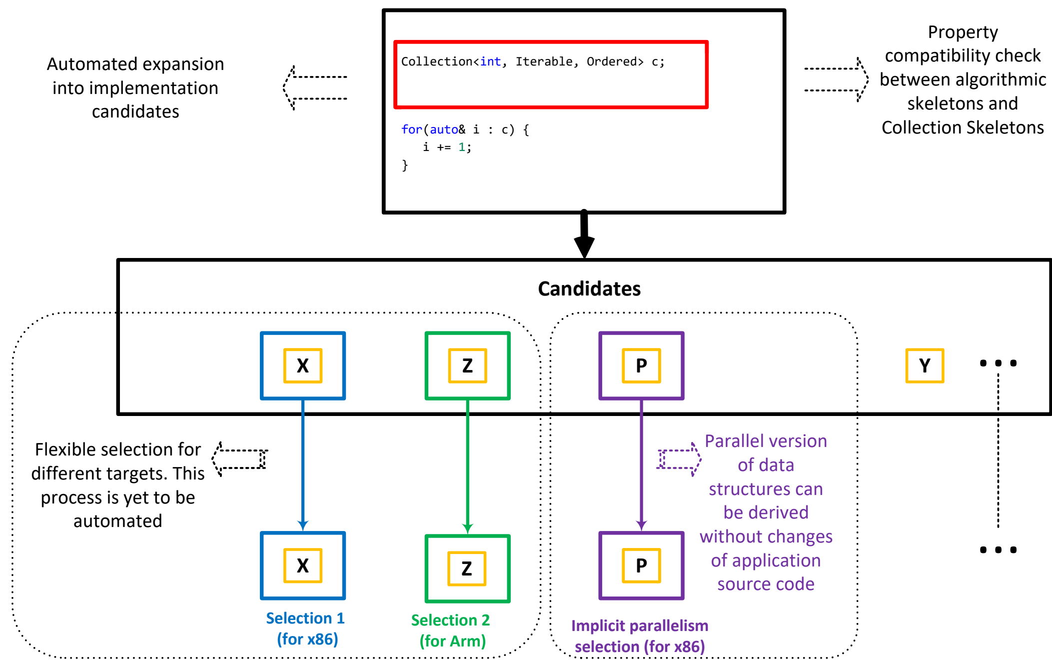


Figure 4.8: Figure 4.7 with more details

model either an individual data structure/ADT or a broader category of data structures. For both ADTs and their specific implementations, i.e., the data structures we have studied, determining the actual structures is achievable when a set of properties is provided. From the programmer's perspective, interacting directly with properties may better align with their needs. By specifying their requirements for a data collection, the end-resulting inferred data structure will hopefully be better performing than selecting one directly from a library. Figure 4.11, Figure 4.12, and Figure 4.13 present the properties identification for a `list` ADT, a `std::list` from the C++ STL, and a `std::list` from the C++ STL with consideration of its operations.

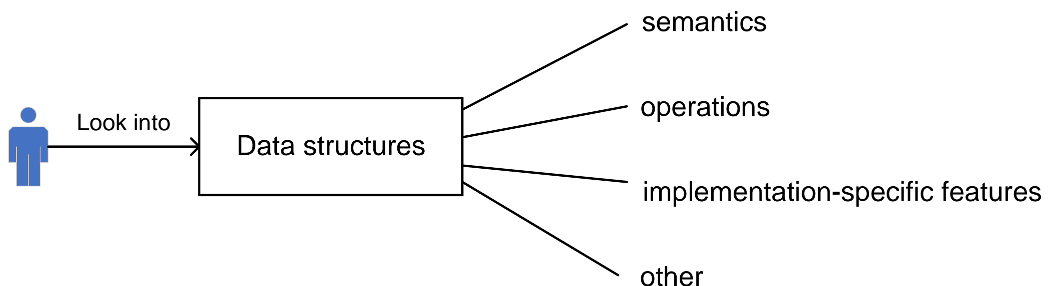


Figure 4.9: Property identification for data structures

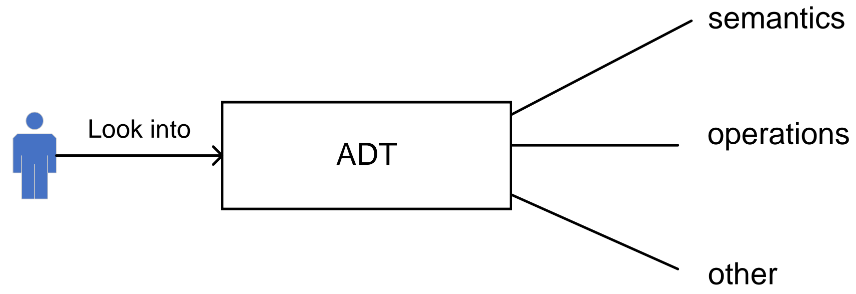


Figure 4.10: Property identification for ADTs

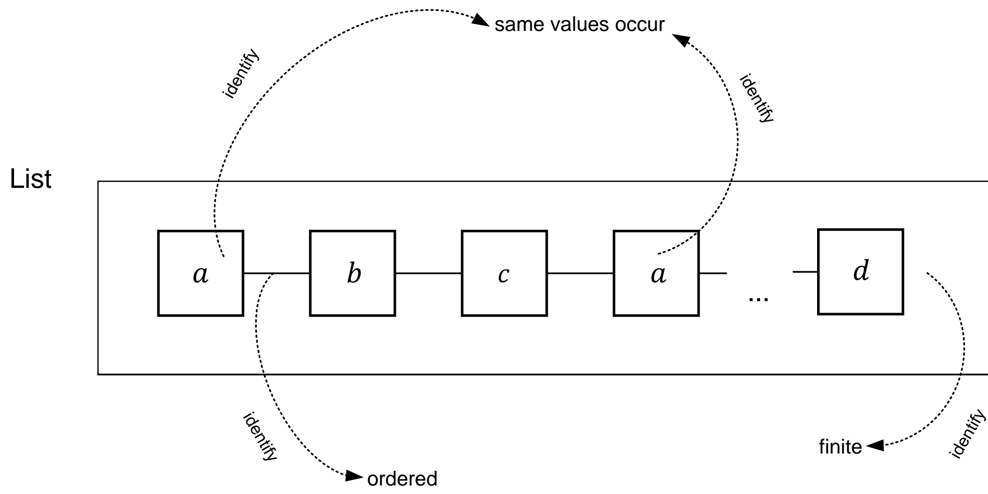


Figure 4.11: Property identification for a List

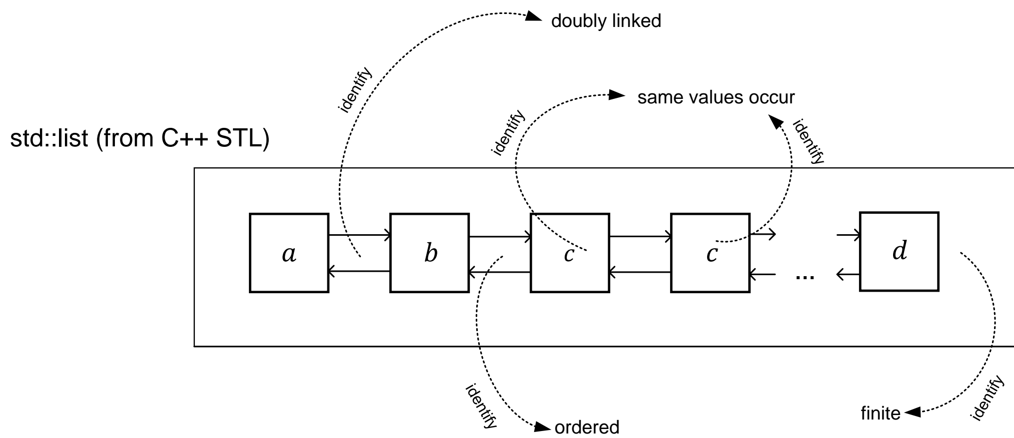


Figure 4.12: Property identification for a std list

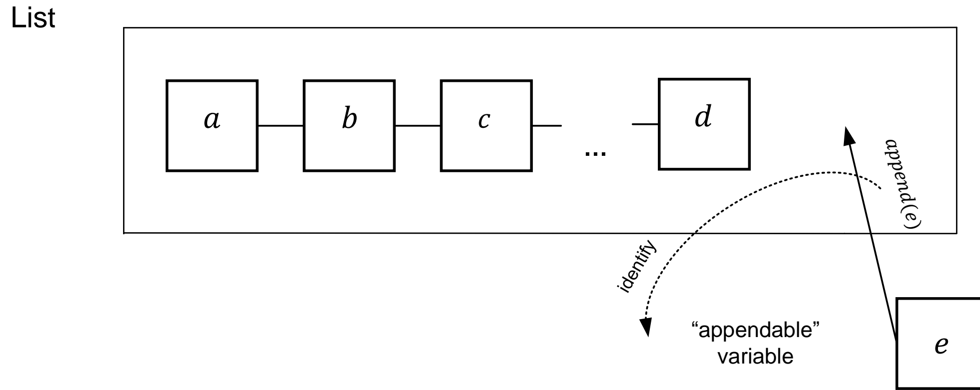


Figure 4.13: Property identification for a List (with operations)

In the following section, we will delve deeper into the properties we have identified with the method just introduced. We also propose definitions for those identified properties to build a property model which can be used by the programmers to declare a data collection.

4.4 Properties for Collection Skeletons

As introduced in the previous section, we identified and exposed the properties by investigating the semantics of the ADTs, exploring the interface functions provided by popular programming languages' standard libraries and analysing other information associated with the standard libraries. Thus, we classify all discovered properties into three categories, which are 1. semantic properties, 2. interface properties and 3. non-functional properties. However, we only introduce semantic properties and interface properties and utilise a restricted number of the properties to help model the Collection Skeletons in our prototype library. Non-functional properties and more properties from semantic properties and interface properties will be integrated to the implementation in future research. In this thesis, we prioritise the most frequently used properties, and thus do not incorporate all the semantic and interface properties explored. The Collection Skeletons have been designed with maximum flexibility in mind, allowing for easy extension with additional properties while maintaining the integrity of the existing prototype library.

Table 4.1: Groups of properties, their definitions and API usage

Semantic property	Definitions	API Parameters
Uniqueness	$\forall x, \forall y \in \mathtt{C} : val(x) = val(y) \Rightarrow x = y$	Unique
	where x, y are elements of collection C and $val(x), val(y)$ are their values	
	otherwise	Duplicate
Circularity	$index(next(last)) = index(first)$	Circular
	where $last = iterator(pos(\vert C\vert - 1))$ and $first = iterator(pos(0))$	
	otherwise	Noncircular
Interface property	Definitions	API Parameters
Size	$\vert\mathtt{C}\vert = size$ where $size$ is a constant	Fixed
	Otherwise (provides e.g., functions **append**, **insert**)	Resizable
Iterable	provides function **iterator** to return a forward iterator	Iterable
	provides function **backward_iterator** to return a backward iterator	Reverse_Iterable
	provides both forward and backward iterator	Bidirectional
	otherwise	Non_Iterable
Accessibility	provides random access functions **get(i)**, **set(i,value)**, **c[i]**	Random
	provides associative access functions **get(key)**, **set(key,value)**, **c[key]**	Associative
Splitability	provides functions **splitAt**, **splitBy** with	
	$splitAt(pos((x_1), pos(x_2), ...pos(x_n)), x_n \in \mathtt{C}$	Nonsplit, SplitN, Splitable
	$splitBy(x_1, x_2, ...x_n), x_n \in \mathtt{C}$	
UnionFind	provides function **union(c1,c2)** and **find(value)** for disjoint collection of collections	UnionFind
Hybrid property	Definitions	API Parameters
Order	Elements can be retrieved in defined insertion order	Ordered
	$push(x) \preceq push(y) \Rightarrow y = pop() \preceq x = pop()$	LIFO
	$enqueue(x) \preceq enqueue(y) \Rightarrow x = dequeue() \preceq y = dequeue()$	FIFO
	$priority(c[0]) > priority(y) \forall y \in \mathtt{C}$ and etc.	Priority_Order
	$x < y \Rightarrow pos(x) < pos(y) \forall x, y \in \mathtt{C}$ for Order by value	OrderByValue
	otherwise	Unordered

4.4.1 Semantic Properties

Semantic properties manipulate the behaviour of the methods with which collections are accessed or modified. For our minimal prototype library we have implemented the following two semantic properties:

Uniqueness A collection only contains unique elements, i.e., there are no two elements with equal value contained in the collection. In Table 4.1, function `val()` represents the value of an element. If elements are not unique, the collection contains duplicates.

Circularity A collection is circular if, as part of an iterator, "wraps around" from the last position of the collection back to its first position, or vice versa.

4.4.2 Interface Properties

Interface properties specify certain functionality, usually in the form of access methods to be provided by the collection. In this work we consider the following interface properties:

Size A collection can be resizable if its size is not statically known at compile time or fixed at the time of its instantiation, i.e., its size can be changed after construction. For example, this could be the result of invoking functions `insert` or `append`; otherwise it is fixed, having an invariant size during its life cycle.

Iterable Elements of an iterable collection can be traversed through means of an iterator, where each element is visited exactly once. This property lets the user specify whether a collection is iterable, and if so, whether forward, backward or bidirectional iterators are provided.

Accessibility This property specifies how elements of a collection can be accessed. We support two access modes, including random access and associative access via the *Random* and *Associative* properties, respectively. In terms of access functions *get* and *set* and the operator `[]` are provided that either take an integer index or a key value as a parameter.

Splitability A collection is splitable if it can be decomposed into two or more collections whose union are the elements of the original collection. We support splitting by index, i.e., elements before and after a given index position, and splitting by value, i.e., elements less or greater than a pivot value. This is encapsulated in the behaviours of functions `splitAt()` and `splitBy()`.

UnionFind This property is defined for collection of collections and provides `union` and `find` operations. Assuming set-like collections this property provides a collection

Table 4.2: Interface properties and their corresponding member functions

Properties		Guaranteed functions	Explanations
Size	Resizable	`void insert(Iter iter,T elem)`	Insert `elem` at position `iter` of a collection Not applicable for Fixed or `const` collections once initialised. “Overloaded” for Collections with different interface properties

Iterable	Iterable	`Iter begin()`
		`Iter end()`
		`Iter next(Iter iter)`
			`begin()` returns an iterator to the beginning of a collection
			`end()` returns an iterator to the end of a collection
			`next()` returns the (immediately) next iterator or element
			of the current iterator (iter)

	Bidirectional	The three above plus
		`Iter prev(Iter iter)`
		`Iter rbegin()`
		`Iter rend()`
			`prev()` returns the (immediately) previous iterator/element
			of the current iterator (iter)
			`rbegin()` returns a backward iterator to the beginning
			`rend()` returns a backward iterator to the end

Accessibility	Random	`Iter operator[](size_t i)`	Access the element through an index
	Associative	`Iter operator[](Key key)`	Access the element through a key
Order	FIFO	`void enqueue(T elem)`	Enqueue element
		`T dequeue()`	Dequeue element
	LIFO	`void push(T elem)`	Push element
		`void T pop()`	Pop element
		`void T top()`	Access top element without popping
	Priority_Order	`void T front()`	Access element with highest priority
UnionFind	UnionFind	`void union(C<T> c1, c2)`	Merge disjoint collections c1 and c2 in a collection of collections
		`C<T> find(T elem)`	Find collection **elem** is element of
Splitability	Splitable	`tuple<Iter> splitAt(Iter iter)`	Split a collection around iterator `iter`
		`tuple<Iter> splitBy(T elem)`	Split a collection around pivot element `elem`

abstraction for disjoint sets and a function for merging collections or identifying the collection containing a specific element.

4.4.3 Hybrid Properties

Some properties are of hybrid nature, i.e., they both specify access methods and also change the way operations behave semantically. An example of such a hybrid property is *order*:

Order This property is an interface property because LIFO or First-In-First-Out (FIFO)

Table 4.3: Default member functions

Default methods	Explanations
`size_t size()`	Return the size of a collection
`bool isEmpty()`	Check if a collection is empty, `true` for empty, otherwise not empty
Collection()	Default Constructor
Collection(size_t size)	Constructor by size
Collection(const Collection& c)	Copy constructor

order provide pop & push and enqueue & dequeue operation pairs, respectively. If an operation *push(x)* immediately preceeds ($\preceq$) an operation *push(y)*, then we expect two subsequent *pop()* operations to first return y and then x. However, order is also a semantic property because the specified order influences the behaviour of the iterator function next. Iterator-based traversals of the collection will yield different traversal orders depending on the specified order (insertion order, ordered by value, LIFO, FIFO, priority queue order, or unordered), thus changing the semantics of the iterator access.

Collection Skeletons provide a convenient and flexible way to define data collections using combinations of the desired properties in a purely declarative way. As of a proof of concept, we integrate the Collection Skeletons with modern C++ programming practice and enable users to transition legacy C/C++ code. We implemented the prototype library including API of the Collection Skeletons using C++ template metaprogramming, where a collection can be defined by a series of template parameters. The evaluation of this prototype library will be discussed in Section 4.6.

4.4.4 Non-functional Properties

Properties that do not fall under either semantic or interface categories are termed non-functional properties, e.g., time and energy [30]. These non-functional properties draw inspiration from the time complexity specifications inherent in the C++ STL. In the STL, containers uphold the time complexity standards for their respective member functions. For instance, the insert function of a vector container exhibits a time complexity in $O(N)$, whereas the push_back function from the same container operates with a time complexity in $O(1)$ for a single operation.

Although we do not currently integrate non-functional properties into our Collection Skeletons and the associated prototype library, the interaction with non-functional properties should be similar to that with semantic properties and interface properties. For example, **Computational Time** can be a non-functional property and a programmer can request a collection has a *fast* find operation. Then, if eligible, a find function with a *fast* feature will be generated to be applied over the collection. And similarly, the implementational details are transparent to the programmer as they only need to specify what properties they would like.

Furthermore, it is possible to perform fine-grained specification with non-functional properties. For example, we characterise the time complexity of an insert function as a non-functional property, e.g., where there can be an O_N *insert* to specify an insert

operation whose time complexity is in $O(N)$. Correspondingly, when such a non-functional property is stipulated, the chosen concrete data structure must conform to this requirement; that is, the insert function of the data collection must demonstrate a time complexity in $O(N)$.

Non-functional properties differ from semantic and interface properties, which are largely determined by the definition of the data collection. In contrast, non-functional properties are intrinsically tied to their implementations. For instance, it is possible to design a `vector` container where the `insert` function exhibits a time complexity in $O(1)$ with some hardware. Additionally, space complexity is also grouped under non-functional properties. Given that non-functional properties are deeply rooted in their specific implementation details as well as the underlying architecture and platforms, we have yet to incorporate them into the prototype library, earmarking them for future exploration.

4.5 Design Principles of the Prototype Library

As suggested by the purpose and definition of Collection Skeletons, we design the prototype library of Collection Skeletons to make it abstract, flexible and portable. We implemented a prototype library for Collection Skeletons with C++ metaprogramming for the purpose of proof-of-concept. Modern C++ offers features such as type traits and template metaprogramming, making it apt for implementing type-related code for compile-time operations. Given the widespread use of C++, numerous benchmark programs are available, facilitating a comprehensive comparison to assess the overhead and performance of the Collection Skeletons. With the compile-time programmability enabled by C++ metaprogramming, programmers can specify a set of properties as template parameter declarations. This approach maintains opportunities for us, as library designers, to enable the prototype library to select and dispatch the most appropriate concrete data structure implementations during compile-time, aligning them with the declared properties.

Based on the proposed semantic and interface properties, we have developed a prototype library for the Collection Skeletons. This library is presented as a parameterized abstract declaration and comprises two primary components:

- **A programming API**, implemented using C++ template metaprogramming, which accepts properties as declarative abstractions. This serves as the frontend

of the programming model.

- A multi-staged, pattern matching-based **mapping mechanism** that bridges the gap between the declarative abstractions and the concrete internal data structures. These internal structures are currently sourced from C++ STL, Boost collections, and other data structure libraries.

As illustrated in Figure 4.2, programmers specify their desired data collection properties via the programming API. The underlying library then determines the appropriate concrete data structures based on these properties. In keeping with our commitment to transparency, the intricacies of this decision-making process are abstracted away from the programmers and are the responsibility of the library developers. While we currently employ a multi-staged pattern matching algorithm for determining the optimal data structure, alternative strategies can also be adopted in future implementations.

Moreover, the two components are decoupled from each other. This means that modifications to either component, such as alterations to the multi-staged pattern matching algorithm, will not affect the other, which is showcased in Figure 4.14. This modular design enhances flexibility and extensibility, which will prove advantageous in subsequent chapters.

It is worthy mentioning that there are also other possible design decisions to fulfill the Collection Skeletons. For example, it is feasible to implement Collection Skeletons with Scala, as Scala has a well-designed type system and powerful metaprogramming. Therefore, implementing the properties as type parameters in Scala should be as straightforward as our approach. Furthermore, Collection Skeletons can be implemented as a DSL, or even a complete programming language, which could potentially improve the expressiveness of Collection Skeletons. Nevertheless, for a proof-of-concept purpose, implementing Collection Skeletons as a template-based C++ library is an optimal choice. With a C++ prototype library, it is convenient to perform a like-for-like comparison over benchmark programs written in legacy C code, which will be showcased in Section 4.6.

4.5.1 API of Collection Skeletons

We employ C++ template metaprogramming in the creation of our Collection Skeletons library. However, in contrast to the C++ STL —which also leverages template metaprogramming —we shield this complexity from the application developers. Our API is designed such that it assists programmers in populating the template class using proper-

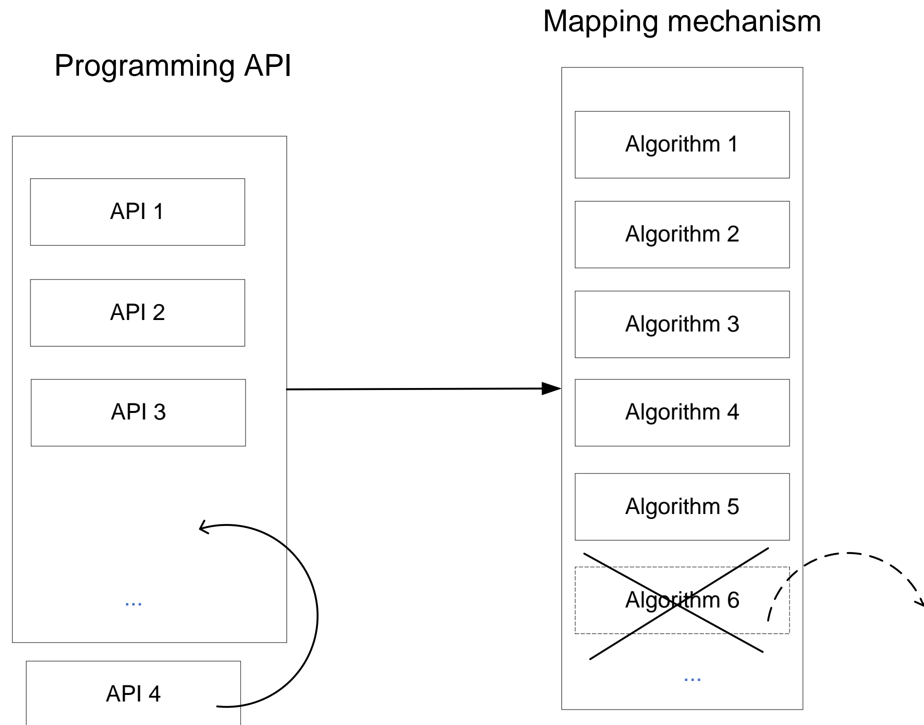


Figure 4.14: Two decoupled components

ties as type parameters. Consequently, users require only a foundational understanding of C++ programming to utilise our API for the prototype library.

Figure 4.15 provides an overview of the Collection interface as provided by our prototype library. In the API, we refer to the property parameters enclosed in the angle brackets as the *property list*. In this API, the first type parameter `T` is the type of the elementary data to be stored by the data collection. Following `T`, P_1, P_2 to P_n are different properties of the desired data collection. These parameters of properties can be found in the column *API Parameters* from Table 4.1.

A special case occurs when the parameter *Fixed* is provided, i.e., the requested collection is *Fixed*; an unsigned integer constant must be provided through a non-type template parameter of type alias `size` as the fixed size of the collection.

Variadic type parameters (`F...`) are optional type parameters that may need to be applied when declaring a data collection, e.g., an *OrderByValue* data collection where the user requests a collection ordered by values as specified by a user-defined comparison function. Such a user-defined comparison function can be passed as input through the variadic type parameters `F...`. Or when an *Associative* collection is requested, the *Value* type should be declared at `F...` where the *Key* type is defined by

`T`.

Table 4.2 summarises interface properties and their corresponding member functions. For convenience, we have defined a set of default methods available to all collections, which are described in Table 4.3.

In addition to directly declaring the data collection using the *property list*, users have the option to employ a type alias for the property-based representation, streamlining subsequent usage. For instance,

```
using C1 = Collection<T,P1,P2,...Pn>
```

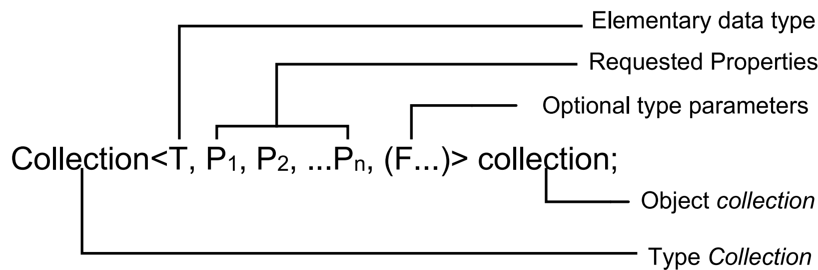


Figure 4.15: Overview of the Collection interface

In this code snippet, `C1` serves as a type alias. By employing this type alias, users can instantiate a data collection in subsequent program sections without the need to repetitively specify the same intricate Collection type. As an illustrative example, this approach enables users to define a *set* alias corresponding to a collection embodying the characteristics of a `set`.

For enhanced user convenience in our prototype library, we default to the assumption that all data collections inherently possess the properties of being *Duplicate*, *Noncircular*, *Iterable*, and *Ordered*. This decision stems from our observation that collections embodying these attributes are more commonly utilised compared to others with divergent characteristics. Thus, unless explicitly indicated to the contrary, these default properties are assumed.

4.5.2 Concrete Implementation Decision —the Flexibility

With the combination of properties provided, the Collection Skeletons can then advance to the subsequent phase: determining the specific data structure for code generation. The chosen combinations of properties need undergo consistency checks and be mapped

to applicable concrete data structures for execution. This objective is achieved through the multi-staged pattern matching algorithm detailed in this section.

With a list of properties as an input according to Figure 4.15, the library employs multi-staged pattern matching on the property list and aims to eventually select a concrete data structure providing those properties. If none of the available implementation data structures satisfy the requested properties, the compilation will be interrupted with a human-readable error message. When a combination of property parameters can be satisfied by at least one concrete implementation, we denote this combination of properties *eligible* and the implementation data structures become *eligible* candidates. An overview of this process is shown in Figure 4.16.

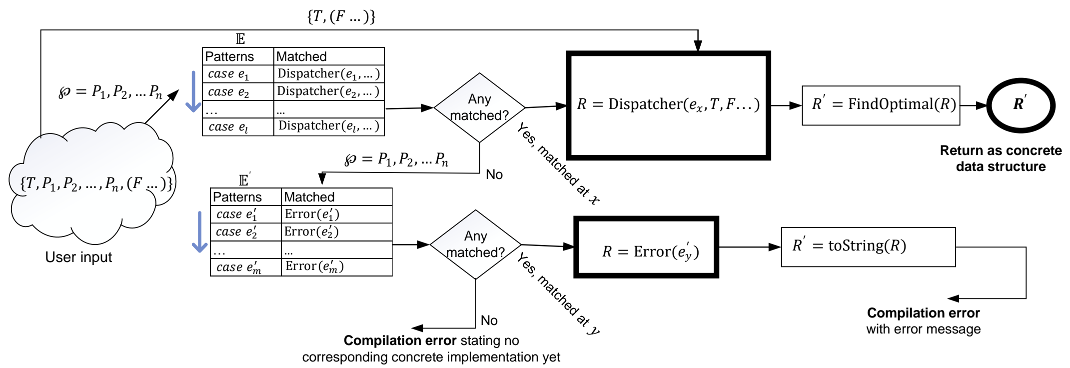


Figure 4.16: Deciding the concrete data structure given an input property-based declaration

Conceptually, this process works akin to a database query —the user-provided input property list is transformed into a *query*, and our library performs a *lookup* over available implementation data structures and their provided properties through pattern matching. If a query can be matched, i.e., the input property list is eligible, the library returns a concrete data structure candidate and continue compilation; otherwise, if the requested properties are found to be internally inconsistent or no suitable implementation data structure that satisfies all of the requested properties can be found, compilation will be interrupted with a human-readable error message.

A detailed description of the pattern matching process is presented in Algorithm 1. Indeed, instead of directly returning the implementing data structure, we incorporate an additional level of indirection via a *dispatcher* class. For scenarios where multiple suitable implementation data structures are identified, we encapsulate the implementation candidates within an *Intermediate Class*. This class is then responsible for selecting and returning one of the viable implementations. Incidentally, this intermediate class con-

cept also provides us with a convenient way of providing extensibility to our Collection Skeletons framework as new data structure implementations can be added conveniently without change of the programming model or any user application code.

For example, a user might request an *Ordered* and *Iterable* collection of integers as follows:

```
Collection<int, Ordered, Iterable> c
```

Both a double-linked and single-linked list meet the requirement, among many other data structures. In fact, in our prototype library, there are three concrete data structure candidates wrapped from `std::list`, `std::forward_list` and `boost::slist` available, which satisfy the requested properties. The prototype library can be easily extended for more candidate concrete data structures as long as they satisfy the properties *Ordered* and *Iterable*. The corresponding structure of the relevant intermediate class is shown in Figure 4.17. This class acts as a type alias for the three template classes associated with macros. With predefined macro configurations passed through the function `FindOptimal`, a single concrete data structure is selected and returned.

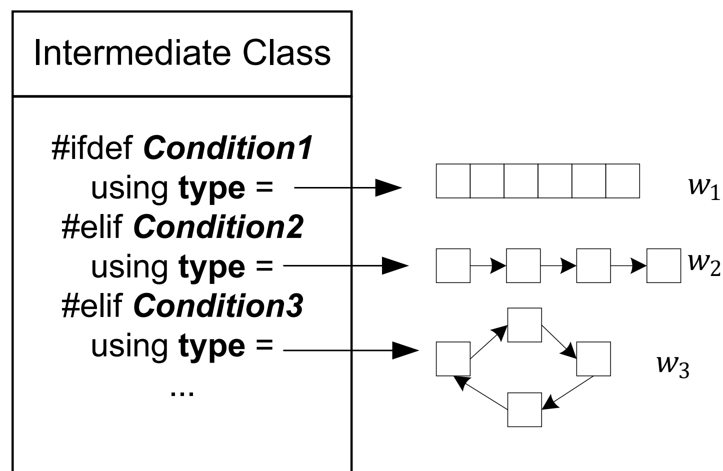


Figure 4.17: Example overview of an intermediate class

The intermediate class as well as the function `FindOptimal` expose chances to pick the most suitable concrete data structure with regarding the program during compile time. In this prototype library, the selection process is developed based on the practical programming experience —when there are more than one candidates available, the

prototype library will always return the more frequently-used one based on our programming experience, e.g., for the case above, when property *Ordered* and *Iterable* are specified, the concrete data structure wrapped from `std::list` will be returned. In the prototype library's implementation, priorities have been designated to the concrete data structures within each *Intermediate Class*.

```
Collection<int, Ordered, Iterable> c  ⇒  std::list<int>
```

In this prototype, the selection of the implementation data structure is implemented using macros and C++ template metaprogramming at compile time. That is, both the `FindOptimal` function and the *Intermediate Class* function during compile time. However, this step could be automated or even dynamically executed at runtime. For instance, we could employ the data collection switch techniques introduced in the CollectionSwitch framework [36]. This approach enables adaptive switching between data collections at runtime, which can offer significant performance enhancements. Furthermore, extending beyond this prototype, the declarative abstractions for data collections can be realised in numerous programming languages, including Java and Scala. It's also conceivable to develop a new programming language dedicated solely to the property-based mechanism for data collections. Incorporating machine learning and AI-driven data structure switching could provide additional capabilities, enabling more optimal selections beyond just compile-time and runtime considerations.

4.5.3 Design of the MSPM Algorithm

This section provides a comprehensive overview of the multi-staged pattern matching algorithm (MSPM algorithm), which plays a pivotal role in determining the concrete implementation or returning an error message, based on the list of properties provided as input. The MSPM algorithm accepts the declarative properties as well as the elementary data type and other optional types parameters as input.

In Algorithm 1, $\mathscr{P}$ is the set of properties **parameters** declared; T is the elementary data type to be stored in the data collection, and $F...$ are optional type parameters that maybe requested by the user. The properties in set $\mathscr{P}$ are unique as redundant properties, if for example, a user requests two *Iterables* for a declaration, they will be merged at an early stage. From line 1 to line 5, pattern matching of the eligible combinations of property parameters is performed, where $\mathbb{E}$ is the set of all eligible patterns of eligible combinations. Each time a pattern e is checked against $\mathscr{P}$. This algorithm then decides whether a pattern matches, i.e., a matching case can be found.

Algorithm 1: Property-based data collection deduction

Input : Properties $\mathscr{P}$=P_1,P_2...P_n,
Elementary data type T,
Other template arguments, $(F...)$

Output : The implementation or error information

```
1  for e ⟵ 𝔼 do
2      if 𝒫 ≡ e then
3          R ⟵ Dispatcher(e,T,F...)
4          R' ⟵ FindOptimal(R)
5          return R'
6  for e' ⟵ 𝔼' do
7      if 𝒫 ≡ e' then
8          R ⟵ Error(e')
9          return compilation error toString(R)
10 return Combination toString(𝒫) not yet supported
```

After a case has been found, function `Dispatcher` in line 3 returns an **intermediate class** R based on the combination e and type parameters T, and F(if any). R stores all the information regarding the properties, types, type traits, and macros that can be used by function `FindOptimal` to help deduct the final concrete data structure. R stores templates for many concrete data structures, and more data structures can be integrated to it without much work. Thus, the Collection Skeletons are flexible as one exact declaration can map to different concrete data structures, and with function `FindOptimal` we can get the optimal underlying data structure. If, however, no match has been found in line 1 to line 5, then the Algorithm 1 will go to the next stage —pattern matching for erroneous declarations.

Expected common user errors such as putting a pair of *twofold symmetrical dichotomy* properties into the same property list, make that declarations ineligible. We have encoded the error combinations in $\mathbb{E}^{'}$ and if a match is found, it means the input declaration cannot be resolved to a concrete data structure, and then a function `Error` generates error information based on the pattern and stores the error information in an intermediate class for error usage, denoted as R again. Subsequently, if there are inconsistencies or mismatches, a compilation error is triggered, accompanied by human-readable information generated by the `toString` function using R.

After pattern matching at the error stage, if no concrete data structure is returned and no compilation error is triggered between lines 6 to 9, this indicates that the combination of properties is valid, but a corresponding concrete data structure can not be identified. For example, in our prototype we have not yet support for a collection that is of fixed size, ordered, and associative. A compilation error will be triggered with a message stating that the input properties are not supported.

4.5.4 Rules of Properties and API Design

We have deliberately kept the programming API simple and user-friendly. However, some rules are applied to prevent non-deterministic behaviour at code generation time, or more specifically during the stage of mapping to concrete data structures.

4.5.4.1 The Property List is Order-Free

Sequences of property parameters in the property list are order-free, for example

```
Collection<T, P1, P2, P3>
```

works exactly as

```
Collection<T, P2, P3, P1>
```

where P_1,P_2, and P_3 are different properties that together with `T` form an effective combination. Any permutation on properties P_1, P_2, and P_3, will eventually resolve to the same result, i.e., the same concrete data structure or the same error message, meaning every property parameter provided in this prototype library is parallel to each other.

4.5.4.2 Mutually Exclusive Properties Cannot Co-Exist in a Property List

Some properties are mutually exclusive, e.g., twofold symmetrical dichotomy properties, where a property list cannot have both at the same time. For example, a collection cannot be *Ordered* and *Unordered* at the same time, nor can a collection allow *duplicate* and *unique* elements at the same time. For example, a collection declared as

```
Collection<int, Duplicate, Unique> c
```

is not permitted. Consequently, if mutually exclusive properties are specified in the property list, such as the aforementioned invalid declaration, the compilation will halt and produce an error message. These conflicting scenarios are defined in the error patterns, as outlined in lines 6 to 9 of Algorithm 1.

4.5.4.3 No Guarantee on Algebraic Operations for Properties

Users might anticipate the ability to execute algebraic operations on properties or property lists. Nonetheless, our library does not support this capability at the moment. Users must ensure consistent usage of collections when they are enhancing functions or defining their own operations on these collections. For instance, when dealing with two collections,

```
Collection<T, P1, P2, P3, P4> c1;
```

and

```
Collection<T1, P5, P6, F> c2;
```

where both declarations are eligible, `T` and T_1 are elementary data types, `F` is an optional template argument. A user might want to define a `merge` function that merges the two collections `c1` and `c2`. At this stage, the library does not govern the resulting type of the merged collection if the original collections comprise different property lists. Eventually, we feel the semantic effects of such operations on distinct collection types should be resolved by the user, who must specify the intended behaviour of such an operation. In our future work, we plan to introduce a recommendation mechanism to help the programmers achieve a more appropriate return data collection.

4.5.5 Implementation of the Pattern Matching Algorithm

We utilise C++ metaprogramming, leveraging template classes and type traits available in modern C++, to implement the multi-staged pattern matching algorithm. Notably, this approach does not require any modifications to the compiler, enhancing the flexibility of our prototype library. As is shown in Section 4.5.3, after receiving the declarative representation from the frontend, the pattern matching algorithm operates on the properties to determine the resulting concrete data structure. The implementation details of the multi-staged pattern matching algorithm will be presented in this section by describing the core source code from the prototype library. The idea of the pattern matching implementation is straightforward —where helper template classes `contains`, `selects` and `when` have been provided to enable the pattern matching —i.e., if a list of declared properties parameters *contains* a specific pattern of properties, then *selects* the concrete data structure or the error message predefined given the condition specified by *when*.

```
template<typename T, typename... Ts>
```

```
struct contains: std::disjunction<std::is_same<T, Ts>...> {};
```

Listing 4.1: Check if a type is contained in a variadic type list

Listing 4.1 presents a helper template class `contains` that operates on type parameters to decide if a given type `T` is contained in the variadic type list `Ts...`. This template class acts as the fundamental helper class for the following source code, enabling pattern matching given the input type parameter against the to-be-compared ones.

```
/*type selects*/
template<typename... Args >
using selects = typename std::disjunction<Args...>::type;

/*type when to decide the conditions*/
template<bool V, typename T>
struct when {
    static constexpr bool value = V;
    using type = T;
};
```

Listing 4.2: Compile time structures for pattern matching

selects<
when<*Condition 1, type 1>,*
when<*Condition 2, type 2>,*
when ...
>

Figure 4.18: Grammar of pattern matching implementation

Similarly, helper template classes `selects` and `when` are provided. Figure 4.18 presents the grammar of the pattern matching implementation. In Listing 4.2, `selects` and `when` are the implementation of a basic pattern matching grammar —*when* the case matches, the corresponding class, i.e., the intermediate class, will be *selected*.

```
struct CollectionTypeDispatcher{
    using type = selects<
        when<contains<Random, F...>::value && !contains<Fixed,
            F...>::value,typename pRnd<T>::type >,
```

```
        when<contains<Associative, F...>::value && !contains<
           Random, F...>::value, typename pSeq<T>::type >,
        ...
        ...
    >
}
```

Listing 4.3: Pattern Matching

Listing 4.3 shows some examples of the patterns and their corresponding intermediate classes. In Listing 4.3, the struct `CollectionTypeDispatcher` *dispatches* the concrete data structure based on the conditions specified by the type parameters. For instance, in the first `when` expression, should the condition evaluate to true —that is, if the variadic types pack includes *Random* and excludes *Fixed* —the intermediate class `pRnd` is then returned as the outcome of the pattern matching process.

```
template<typename T>
struct pRnd{
#ifdef WRAPPERSSTL
    using type = wvector;
#else
    ...
#endif
};
```

Listing 4.4: Intermediate Class

Listing 4.4 provides a detailed representation of an intermediate class, further elucidating Figure 4.17 discussed earlier. We also implement pattern matching for incompatible combinations of properties in a similar manner. However, error messages are encapsulated within their respective intermediate classes. For valid combinations that our library does not currently support, we utilise `static_assert` to halt the compilation process, subsequently providing feedback to the user, as outlined in Algorithm 1.

The Collection Skeletons prototype library is fully implemented with C++ and operates at the compile stage, meaning it can be flexible and efficient. We draw on the C++ STL, Boost and other data structure libraries for our pool of concrete data structure implementations. Through the provided wrapping mechanism, it is convenient to extend our library with additional implementations, whilst introducing only minimal runtime overhead. As will be introduced in Chapter 5 and Chapter 6, we extend the library with

parallel data structures from different libraries for parallelism uncovered in alignment with the strategy mentioned above.

4.5.6 Discussion on Alternative Designs

As introduced earlier at Section 4.5, there are several possible alternative designs to implement the Collection Skeletons, for example, we can implement the API as a totally different interface to distinguish the specification on semantic properties and interface properties as follows:

```
Collection<T, Properties_S, Properties_I, (F...)> c
```

Where `Properties_S` is the set of semantic properties, and `Properties_I` is the set of interface properties. For this alternative API implementation, the programmer needs to firstly specify semantic properties and then move to interface properties. By ordering the two categories of properties, the programmer can have a better mindset on what they are specifying and expecting from the library, as they can firstly consider the behaviours of the collection and then the functionality of it. Compared to the chosen order-free one as shown in Section 4.5.4, this alternative design can help the programmer smoothly suit the idea of Collection Skeletons. However, as there are hybrid properties in Collection Skeletons, such strategy might introduce extra ambiguity.

4.6 Evaluation

We conducted an evaluation of our Collection Skeletons prototype library using benchmark suites including Olden [23], ising [137], Rosetta [33], Rodinia [26], Parboil [129], and various open-source projects including libactor [51], tinn [93], shor [118], simpleHash [88], mp3 [142], scheduler [95], md5 [76], joseph [12], infix [46] and kruskal [1], to assess its performance and potential overheads. The majority of the selected benchmark programs have been developed using legacy C code, which indicates the possibility for optimisation through transformation into contemporary C++ code. These benchmarks exhibit a variety of data structures, including those that are pointer-chasing, hence presenting considerable potential for data structure misspecification. We manually adapted these legacy applications to integrate our Collection Skeletons, allowing us to evaluate our system in realistic scenarios. Our primary focus was to gauge the performance differences that might arise due to our innovative abstractions, achieved through a series of before-and-after experiments. In this section, we only evaluate the

sequential performance of the benchmark programs and do not attempt do parallelize them. We will discuss parallelisation in Chapter 5 and Chapter 6.

For the purpose of this evaluation we have manually rewritten the legacy benchmarks written in C, but have replaced the low-level user-defined data structures and their access functions with their equivalent collections from our framework. No other code rewriting has been performed and the same input data have been used to facilitate a like-for-like comparison. Details of the benchmarks can be found in Table 4.4, where we have manually identified the original data structures from the benchmark programs and their properties from the problem domain. We intentionally overlook any opportunity for parallelisation or optimisation that arose during the rewriting phase. Our focus for this chapter is strictly on evaluating the fundamental and sequential use cases. We ensure the evaluation is a like-for-like evaluation rather than leveraging the manually rewriting.

Upon extracting the properties and manually analysing the use of data structures in each benchmark program, we substituted the original data structures with their declarative counterparts from our library. Column *Replaced* are concrete data structures before wrapped and obtained by manually checking the declaration and the pattern matching rules from the prototype library. More details of the replaced declarative counterparts and their concrete data structures that would be deducted by the library are also presented in Table 4.4. In column *Replaced (concrete)*, `list`, `set`, `vector`, `map`, `queue`, `stack`, and `priority_queue` are from the C++ STL, `circular list` and `disjoint_set` are from the Boost library.

We use Clang 12 and Clang++ 12 for compilation of the original source code and rewritten programs of the benchmarks separately, along with the "-O2" compiler flags. Besides, we enable C++ 17 compiling by setting "-std=c++17" for Clang++ 12. Experiments have been performed on a 6-core Intel 10710U desktop with 12 threads 16GB memory (Intel Desktop), a 2×18-core Intel Gold 6154 server with 72 threads and 768GB memory (Intel Server), and a 4-core Oracle Cloud Ampere A1 Compute server with 4 threads and 24GB memory (Arm Server), respectively. For evaluation in this chapter, we have disabled Simultaneous multithreading (SMT) for all platforms to ensure the programs in sequential mode. For measuring computation time more precisely, all the programs have been run 5 times to get the average values.

Table 4.4: List of the benchmarks, their data structures, extracted properties and replaced implementations

Benchmark	Source	Original Data structures	Extracted properties from problem domain	Replaced (concrete)
treeadd	Olden [23]	balanced binary tree	Split2	tree2
ising	Github [137]	linked list	Iterable	list
set	Rosetta [33]	linked list	Unique	set
libactor	Github [51]	linked list	Iterable	list
tinn	Github [93]	array	Random	vector
shor	Github [118]	linked list array	Random	vector
simpleHash	Github [88]	linked list (of linked lists)	Random, Iterable	vector (of lists)
mp3	Github [142]	linked list	Unique	set
scheduler	Github [95]	min heap	Priority_Order, OrderByValue	priority queue
md5	Github [76]	hashmap	OrderByValue, Associative, Unordered	map
bisort	Olden [23]	binary tree	Split2	tree2
lud	Rodinia [26]	array	Random	vector
kmeans	Rodinia [26]	array	Random	vector
mri-q	Parboil [129]	array	Random	vector
joseph	Github [12]	circular linked list	Circular, Iterable	circular list
infix	Github [46]	queue stack	Non_Iterable, FIFO Non_Iterable, LIFO	queue stack
kruskal	Geeksforgeeks [1]	map and array	UnionFind	disjoint_set

4.6.1 Experimental Results

4.6.1.1 Overall Performance

Figure 4.19 presents the results of the computational performance variation (speedup) for the 17 benchmarks. A speedup greater than 1.0 indicates that the programs rewritten using Collection Skeletons exhibit a performance improvement compared to the original versions; conversely, values less than 1.0 suggest a decline in computational performance.

From Figure 4.19, we can see that most of the replaced benchmarks have similar or better computational performance than the baseline. It should be noted that simply porting a C program to its C++ equivalent would never achieve such a significant performance improvement. Some replaced benchmarks have much better performance than the baseline ones, suggesting that for some benchmarks, by simply replacing the data structures of the original benchmarks, the performance can be greatly improved. For some benchmarks, the speedup is not substantial, e.g., the replaced ising benchmark only ran a factor of 1.04 faster than the baseline. Although the ising is a small benchmark and does not consist of complex structures, the speedup is still important to be recognised. There are only a few benchmark programs reporting a performance decrease compared to the baseline. For example, in benchmark mri-q, where highly optimised arrays associated with manual memory management have been used in the baseline program, it is understandable that our version using Collection Skeletons did not outperform the original one.

There are differences in speedup across the three architectures, e.g., for the *joseph* benchmark which represents a best case scenario, the speedup on the three architectures is 11.20, 16.37, and 10.36. The great performance boost reported in the joseph benchmark could be the replaced circular data structure is a much better implementation compared to that in the original benchmark. Nearly half of the benchmark programs report a speedup between a factor of 1.00 to 2.00, and others report greater performance boost. In general, the overall results show that the replaced versions have similar or better computational performance for almost every benchmark on the three architectures. This confirms that our Collection Skeletons library can increase the performance for almost all the benchmarks by replacing the original data structures that were misspecified or overspecified with those hidden behind property-based declarations.

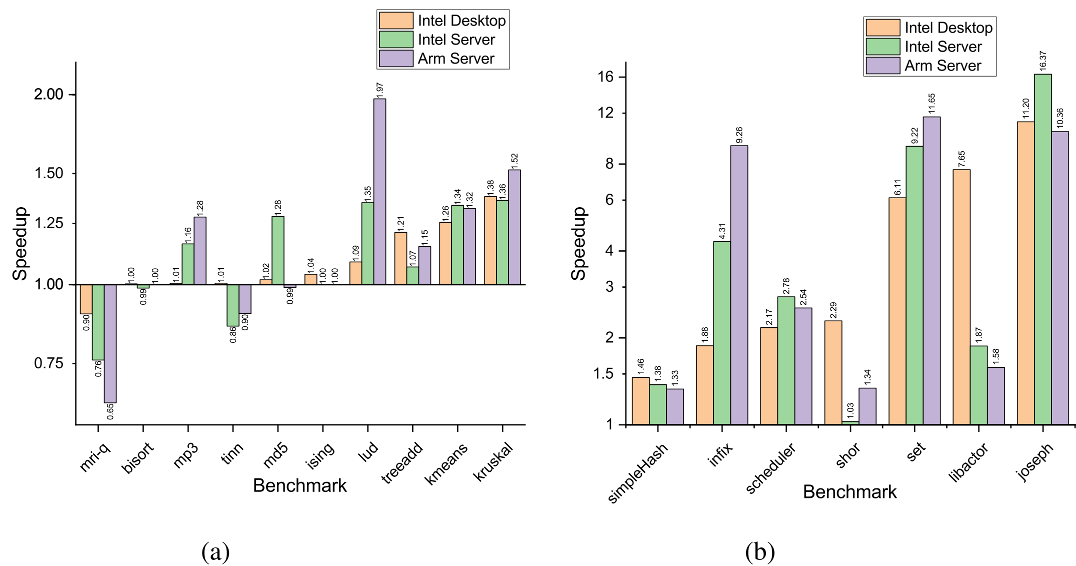


Figure 4.19: Speedup of the benchmarks on three platforms

4.6.1.2 Implementation Flexibility

As mentioned in Chapter 4, a single collection with a property list can lead to more than one concrete data structure as candidates. Here we have selected those benchmarks where this is the case, and we have evaluated the computational performance for each of the feasible concrete data structures. We have found that benchmark treeadd, bisort, ising, set, libactor, tinn, shor, simpleHash, mp3, lud, kmeans and mri-q can be implemented using different concrete data structures from our library. As the mapping and deduction process is transparent to the user, there is no need to modify any user code. We only need to modify some compile-time macros to alter the behaviour of the function `FindOptimal` operating on intermediate classes to *swap out* the concrete implementations. We conducted the experiments using the method outlined at the beginning of this section on the same three platforms, and the results are showcased in Table 4.5.

From Table 4.5, we can conclude that for different target platforms, the optimal concrete implementations can be different, e.g., slist from the Boost library is the optimal data structure on the Intel Desktop, while list from STL is the optimal data structure on the Intel Server. Furthermore, for some benchmarks, e.g., treeadd and bisort, the default chosen collection is a binary tree implemented through pointers, which is the de facto standard way; however, the array-based binary tree has better speedup according to Table 4.5.

Table 4.5: Optimal implementations for selected benchmarks

Benchmark	Intel Desktop		Intel Server		Arm Server	
	Optimal	Speedup	Optimal	Speedup	Optimal	Speedup
treeadd	array_tree	1.61	array_tree	1.09	array_tree	6.37
bisort	array_tree	1.21	array_tree	1.33	array_tree	1.54
ising	slist	1.08	list	1.00	list	1.00
set	unordered_set	6.09	unordered_set	9.22	unordered_set	11.65
libactor	list	7.65	list	1.87	list	1.58
tinn	vector	1.01	vector	0.88	vector	0.9
shor	vector	2.29	vector	1.23	vector	1.34
simpleHash	forward_list	1.61	forward_list	1.44	forward_list	1.36
mp3	flat_set	1.01	set	1.16	set	1.28
lud	vector	1.09	vector	1.34	vector	1.97
kmeans	vector	1.26	vector	1.34	vector	1.32
mri-q	vector	0.90	vector	0.76	vector	0.65

The experimental results confirm that our prototype library offers implementation flexibility, allowing a single declaration to map to multiple concrete data structures based on the combination of specified properties. This decouples the programmer's intent from specific data structure implementations, providing both abstraction and performance.

In our future research, we envision a scenario where a single piece of code allows the Collection Skeletons to autonomously select the best-suited concrete data structures and the optimal platform to execute the program for better computational performance. In this paradigm, programmers only need to define data collections based on their desired properties, transparently streamlining development and potentially boosting performance across various platforms.

4.6.1.3 Overhead

The prototype library has been implemented with C++ metaprogramming, meaning the process of concrete data structure deduction and substitution happens during compile time, which should introduce minimum-to-no extra overhead at runtime [113]. However, the compiler might optimise the code differently when it evaluates the encapsulated data structures than the concrete ones. Thus, to evaluate the overhead of the prototype library, we performed a back-to-back experiment to compare the computational time of the benchmark rewritten with the property-based declarations and that rewritten directly

with the concrete implementations according to our prototype library. To replace the property-based declarations with their concrete implementations, we manually checked with the deduction rules to evaluate the concrete data structures. Based on the same software and hardware environment as described earlier this chapter, we ran the ising benchmark 10 times for both configurations, and compared the mean and variance.

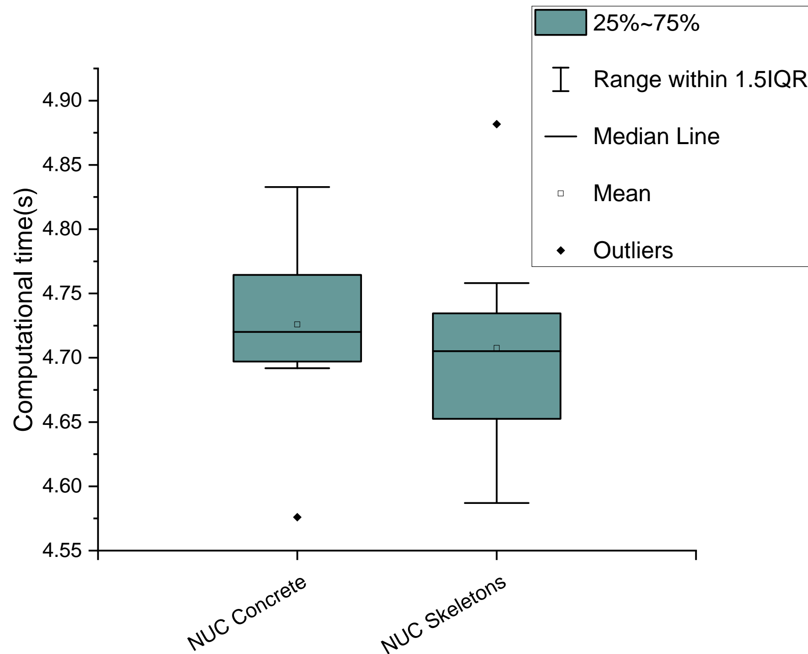


Figure 4.20: Computational time for Collection Skeletons versus concrete data structures on Intel Desktop

Figure 4.20, Figure 4.21, and Figure 4.22 present the box graphs for the computational time for the ising benchmark rewritten with the Collection Skeletons and that rewritten with concrete data structures, which are `std::list` in this case. From these figures it is clear that the computational time on the three platforms are almost identical, especially for Arm Server and Intel Server where the mean, median, and range are extremely close. The result should also hold for other benchmark programs as discussed earlier, as they have all been rewritten with the same strategy.

4.6.1.4 Runtime Performance Influencing Factors

There are several factors that can influence the runtime performance of a program. The size of a data collection, specifically the number of elements it contains, can significantly influence its performance. Additionally, different architectures might present varying

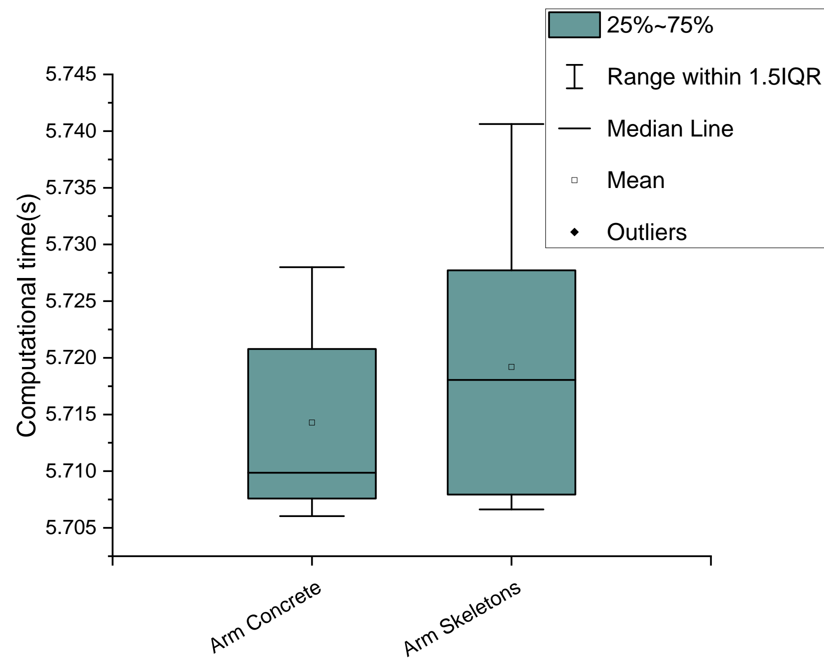


Figure 4.21: Computational time for Collection Skeletons versus concrete data structures on Arm Server

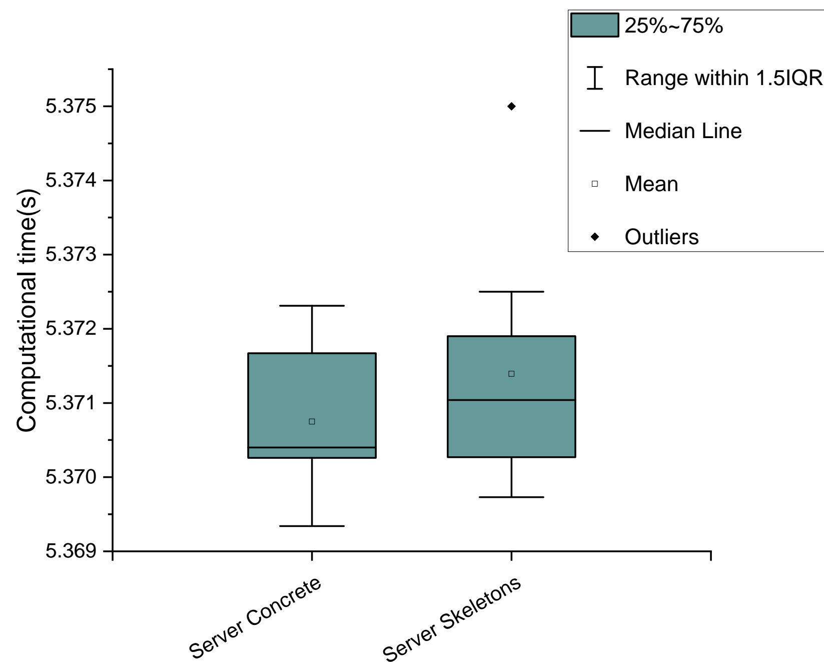


Figure 4.22: Computational time for Collection Skeletons versus concrete data structures on Intel Server

performance trade-offs.

To evaluate the performance variation between different concrete data structure implementations of various size, we pick the *bisort* benchmark for further investigation. During the experiments before, we observed the speedup difference between programs with pointer-based binary tree and array-based binary tree for *bisort*, and it is straightforward to alter the input data size for this benchmark by simply input a different value of the data size upon execution. Apart from altering the input data size, we executed the benchmark programs with both implementations using the same methodology and platforms as outlined at the beginning of this section. The experimental results are presented in Figure 4.23.

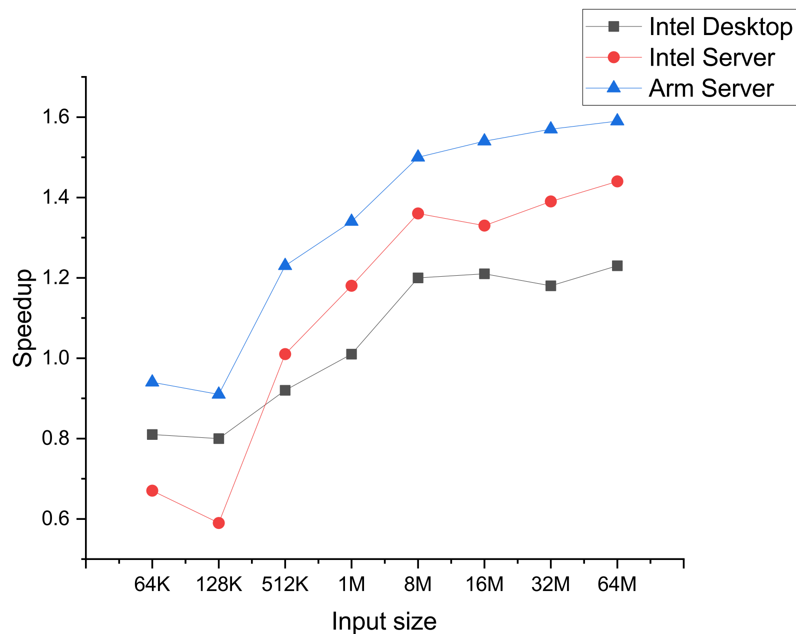


Figure 4.23: Speedup by array-based binary tree for bisort regarding the input size

As depicted in Figure 4.23, we varied the input data size in the range of 64K to 64M for the three platforms. For the *bisort* benchmark, the program accepts an input from the user specifying the size of the random container to be generated. Consequently, the data collection size is determined only at runtime, obviating the need for any modifications to the source code. We consider the original pointer-based implementation (`tree2`) as the baseline against which we assess the relative performance of the array-based binary tree (`array_tree2`). For collections with up to approximately 512k elements, we note that the array-based binary tree lags behind the original pointer-based implementation. However, for larger size of collections, the array-based implementations surpass the

original by margins of up to 60%. In our experiments, we manually determined the final implementation by passing various macros to the `FindOptimal` function. In future work, an automated approach like "CollectionSwitch" [36] could be utilized to streamline this process. By analysing the optimal concrete data structures given a series of compile time and runtime parameters, the original concrete data structure can be swapped out with a more optimal one during runtime. This process should remain transparent to users while enhancing computational and storage performance.

4.7 Chapter Summary

In this section, we propose high-level abstractions for data collections —the Collections Skeletons. These abstractions offer a clean and intuitive interface, making it accessible even to non-specialised programmers who can easily specify data collections with the desired properties, without the need for in-depth knowledge of their implementations. We have prototyped the Collection Skeletons using C++ metaprogramming and evaluated the prototype library with 17 benchmark programs in pure sequential cases. The evaluation results indicate that Collections Skeletons do not introduce overhead; instead, they reveal opportunities for performance improvement in cases where data structures might have been overly specified.

There are still limitations regarding the prototyped library of Collection Skeletons, e.g., non-functional properties are not yet supported. Furthermore, the MSPM algorithm only considers factors that are available during compile time thus it cannot decide a concrete data structure based on runtime factors.

In the next chapter, we will explore the parallelization opportunities in the context of the Collection Skeletons and the prototype using implicit parallelism based on the sequential version introduced in this chapter.

Chapter 5

Exploiting Implicit Parallelism with Collection Skeletons

This chapter defines and explores **Implicit Parallelism** for Collection Skeletons, which can be realised through the integration of concurrent data structures combined with parallelised access methods. It is desirable that the programmers continue to interact with Collection Skeletons as before, and the Collection Skeletons provides concurrent/-parallel implementations when appropriate. Similar to Chapter 4, the concurrent data structures and their associated parallel access functions are hidden from the programmers, thereby enabling the programmers to develop a parallel program without knowing the implementation details, or even being aware of the fact of parallel implementation is exploited.

For a proof-of-concept implementation, we extend our prototype library with concurrent thread-safe data structures, which enables us to safely utilise parallelised operations on these data structures. Based on that, we have further introduced OpenMP directives, which are intended to explore parallelism beyond implicit parallelism provided by concurrent data structures. We evaluated the extended prototype library against the same 17 benchmarks using the same methodology as introduced in Section 4.6, where we firstly substitute the candidates with concurrent data structures only and then insert OpenMP directives around for loops associated with the data collections. While we discuss parallelised access functions for Implicit Parallelism below, these have not been implemented nor evaluated in our prototype library.

5.1 Introduction to Implicit Parallelism

As mentioned earlier in this thesis, writing parallel code or transforming sequential code into a parallel version are not trivial tasks. Numerous techniques aim to provide a portable and flexible parallel programming model, including automatic parallelisation, parallel algorithmic skeletons, and directive-based parallel programming frameworks such as OpenMP and OpenCL. Nevertheless, a gap persists between code developed by programmers and that capable of harnessing the parallelism potential of the underlying hardware [145, 34, 121].

As introduced in Chapter 4, Collection Skeletons offer a solution to address the problem of overspecification of data structures by programmers. Collection Skeletons allow programmers to specify the properties they desire for the data collections rather than dictating the actual data structures for data storage. Since the mapping from property-based declarations to concrete data structures is entirely transparent to non-specialised programmers, it opens up opportunities for Collection Skeletons to select and substitute data structures for improved performance. Additionally, it empowers library programmers to enhance their libraries, such as data structures libraries for optimisations in specific domains, without concerns about compatibility issues with client code developed by non-specialised programmers.

As a result, the opportunity for parallelisation based on Collection Skeletons can be revealed —integrating concurrent data structures together with parallel access functions is just as manageable in terms of effort as integrating sequential data structures as concrete implementations. Thus, based on Collection Skeletons, which are designed to be flexible, concurrent data structure libraries can be integrated as candidates for the multi-staged pattern matching algorithm introduced in Algorithm 1. From the perspective of Collection Skeletons, when it is available, parallelism from the concurrent data structure together with the parallel access function can be seamlessly provided to programmers, and this entire process remains transparent to them. In fact, programmers may not even need to be aware that they are writing a parallel program, as the Collection Skeletons can implicitly dispatch their property-based declarations to a parallel-friendly or concurrent concrete data structure when feasible.

Furthermore, the transparent data structure dispatch also opens up more opportunities for programmers developing parallel programs. By seamlessly introducing concurrent, thread-safe data structures as candidate concrete data structures for Collection Skeletons, programmers can create parallel, multi-threaded programs without

worrying about potential nondeterministic issues when declaring data collections with specific properties. As a result, other parallel programming techniques beyond implicit parallelism, e.g., OpenMP and the thread library from the STL, can be combined with Implicit Parallelism from the Collection Skeletons.

Definition 1. ***Implicit Parallelism of Collection Skeletons —*** *Implicit Parallelism specifies that the parallel computing is transparent to the* ***users*** *of Collection Skeletons, where the parallelism is hidden by Collection Skeletons through concurrent data structures and parallel operations on those data structures.*

Therefore, we propose use of **Implicit Parallelism** to extend our Collection Skeletons. Beyond and based on Implicit Parallelism, more parallel programming techniques can be integrated more consistently because of the concurrent data structures. Interaction with implicit parallelism of Collection Skeletons is the same as that of Chapter 4; the application programmer specifies and uses data collections with the properties, but Collection Skeletons select concurrent data structures and parallel access functions as the concrete implementation when appropriate. This implies that application programmers do not need to concern themselves with this level of parallelism. It appears to them as the merely an implementation for Collection Skeletons, with the invisible distinction that parallelism could be internally exploited in the concrete data structure with the associated access functions.

5.2 Providing Implicit Parallelism with Collection Skeletons

Collection Skeletons with implicit parallelism should maintain the interaction as a highly abstract programming model for the programmers. Based on Chapter 4, the Collection Skeletons can be extended with concurrent data structures, as merely another choice of concrete data structure. This extension retains the same API, properties, and deduction algorithm, allowing programmers to harness potential parallelism without necessarily being aware that they are writing parallel code.

For example, a programmer requests a data collection with the *Random* property to store integers. Then, they would like to let every element of the collection be incremented. Based on Collection Skeletons introduced in Chapter 4, they can develop the segment of program as,

```
Collection<int, Random> c;
... //initialisation of c
for (auto&i : c){
    i += 1;
}
```

which is the same way of interacting with Collection Skeletons as in Chapter 4. However, this time Implicit Parallelism can be provided in several forms, e.g., where a concrete implementation is available, the Collection Skeletons generate a vector that is suitable for parallel access functions as the concrete data structure, and that is highly parallelised regarding initialisation, which is showcased in Figure 5.1; or, the concrete data structure can be another vector that internally enables the parallel operations (e.g., `access` and `add`) in the for loop, which is shown in Figure 5.2. The eventual performance depends on the concrete data structure. However, from the view of the programmer, they do not really notice any parallelism. Ideally, as discussed in Chapter 4, there can be a smarter algorithm to decide the concrete data structure for optimal performance and even swap out the data structure depending on the workload during runtime.

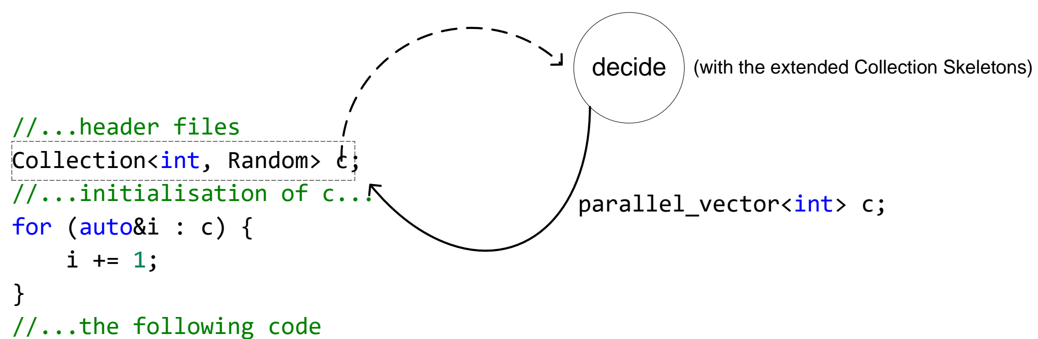


Figure 5.1: A parallel vector substitutes the properties declaration

Collection Skeletons with implicit parallelism integrated can substantially help the non-specialised programmers, as they do not need to make extensive modifications to their code, which is often required in common parallel programming paradigms such as MPI. Furthermore, non-specialised programmers will not encounter compatibility issues with automatic parallel toolkits when programming with Collection Skeletons. Collection Skeletons, by shielding the concrete data structures from programmers, can also conceal the parallelism from them.

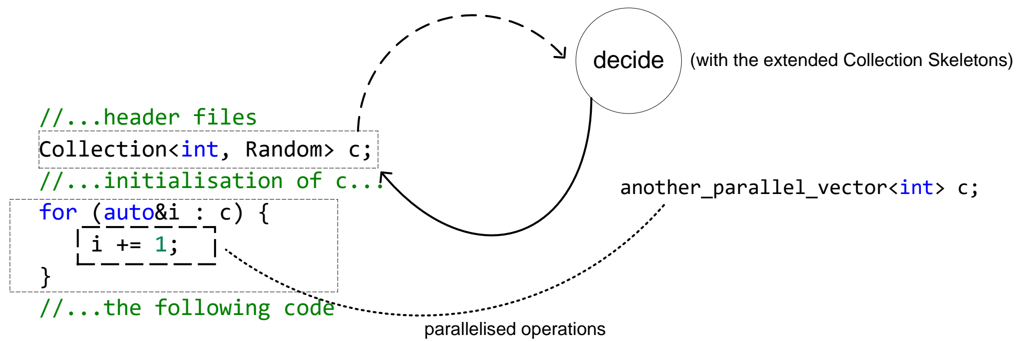


Figure 5.2: Another parallel vector substitutes the properties declaration

5.2.1 Integration with Collection Skeletons

Figure 5.3 presents the overview of the Collection Skeletons with the extension of Implicit Parallelism, where the left of the figure is nearly identical to Figure 4.2 presented in Chapter 4, except there are concurrent data structures and their parallel access functions as additional implementation options at the right of the figure. The programmers specify data collections with properties as in Chapter 4, the Collection Skeletons apply the same algorithm to decide the concrete data structure, but this time it has a different preference where possible a parallel-friendly or concurrent data structure should be returned. In this way, parallelism is hidden from the programmers as just another implementation option by the Collection Skeletons.

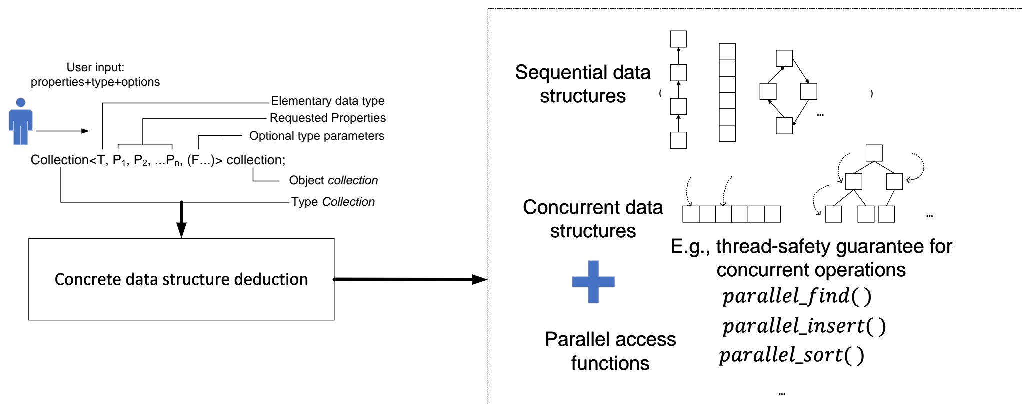


Figure 5.3: Overview of the Collection Skeletons with extension of Implicit Parallelism

Apart from including concurrent data structures as more implementation options alongside the sequential ones, there should be only minor adjustment on the deduction

preference. For example, as implemented in the prototype library in Chapter 4, we can allocate concurrent data structures with more priority than the sequential ones. Similarly, there could be an algorithm that selects the concrete data structures in a smart way with the help of runtime machine learning on multiple factors.

5.2.2 Performance Boost by Implicit Parallelism

Performance boost by Implicit Parallelism can be expected from different aspects,

- For concurrent data structures, they offer preparations for further parallelisation, e.g., guarantees of thread safety —that is, operations on the data structure are thread-safe, which opens up further opportunities for developing parallel programs without concerns over nondeterministic behaviours.

- For parallel access methods, e.g., parallel find and parallel sort, they offer inherent parallelism and therefore enable programmers to harness the multiple threads from the hardware.

Introducing parallelised data structures as additional concrete data structure candidates is as straightforward as sequential data structures for Chapter 4. The listing below of a program presents an example of trying to find whether an integer `x` is in a collection of *Random* property.

```
Collection<int, Random> c;
... //initialisation of c
int x;
c.find(x);  //trying to find if there is an integer x in c
    ;
}
```

With Implicit Parallelism enabled Collection Skeletons, it is feasible that `Collection c` is a concurrent container. Therefore, comparing to its sequential counterpart, there can be a parallel version of function `find` that performs the *find* operation with multiple threads. Depending on multiple factors including hardware properties, the performance can be improved with the parallel `find`.

It can still be profitable when the Implicit Parallelism with Collection Skeletons is realised only with concurrent data structures as concrete data structures. Although concurrent data structures do not provide parallelised access functions, they provide thread safe member functions that preserve thread safety for the data structures when

executing concurrently by multiple threads. For example, a concurrent vector from Intel oneTBB is a concurrent thread safe data structure which provides similar member functions and data member as vectors from other C++ STL implementations. The critical feature of the concurrent vector is that most of its operations are thread safe; for example, it allows multiple threads to perform `push_back` operations while preserving the thread safety of itself.

5.2.3 Beyond Implicit Parallelism

Simply applying concurrent data structures to store data and allow operations upon them might not increase much the performance compared to the sequential data structures. Instead, those concurrent data structures provide **preparation** for further parallel computing —e.g., as the data structure is thread safe, there is no more need to consider the determinism when trying to apply multiple threads operations on the data structure. There are multiple ways to implement parallelism beyond Implicit Parallelism based on its concurrent data structures, for example, directive-based parallel technique such as OpenMP and OpenACC can be easily inserted, or even generated by an auto-parallelizing compiler, to leverage the potential parallelism together with concurrent data structures.

The following code listing presents the same example as before. We assume the collection `c` to have been deducted as a concurrent vector, e.g., a concurrent vector from Intel oneTBB.

```
Collection<int, Random> c;
... //initialisation of c
for (int i = 0; i < n; ++i){  //n is the size of c
    c[i] +=1;
}
```

To further parallelise the segment of code, it is possible to introduce directive-based parallel frameworks and insert the directives. For example, the above code can be further parallelised as,

```
Collection<int, Random> c;
... //initialisation of c
#pragma omp parallel for
for (int i = 0; i < n; ++i){  //n is the size of c
    c[i] +=1;
```

```
}
```

OpenMP's directives can be inserted with the help of external tools such as [126], which can also be transparent to the programmers. As before, the programmers only need to request the data collections with the properties, and the Collection Skeletons library can generate the thread safe data structures and parallel operations that perform upon the data structures. In this way, the programmers can develop parallel code in a more convenient way compared to the traditional methods. While we have shown above that OpenMP could be automatically inserted to introduce implicit parallelism, this would also apply to other similar parallel frameworks.

5.3 Implementation —Prototype Library Extension

Building upon the extension of Collection Skeletons with Implicit Parallelism, we have also expanded the prototype library beyond sequential computing. Following the principles that Implicit parallelism disguises itself from the user of Collection Skeletons as yet another concrete implementation of a data collection with specific properties, we keep the C++ metaprogramming implementation of the prototype and have made minor modifications to include concurrent data structures as concrete candidates. Thus, when developing with the extended library, the programmers define a data collection with exactly the same properties they would use as introduced in Chapter 4. Under control of the revised multi-staged pattern matching algorithm, our library selects a concurrent data structure. As a result, no changes to the application source code are required.

However, at present, the prototype library is only extended with concurrent data structures that guarantee thread safety as the additional concrete data structures. It should be noted that in our extended prototype library, the concurrent data structures are essentially equipped with sequential access functions thus they are not internally parallel. We have not yet implemented parallel access methods upon those concurrent data structures, and thus these methods remain to be subject to our future research.

5.3.1 Concurrent Data Structures in the Collection Skeletons Framework

To extend the prototype library with concurrent containers, we wrapped concurrent containers from Intel oneTBB and Boost. This subsection introduces the concurrent

data structures that we have integrated with the prototype libraries for Implicit Parallelism. All the introduced concurrent data structures have their corresponding sequential counterparts as introduced in Section 4.6.

`cvector`: Concurrent vector wrapped from `concurrent_vector` of Intel oneTBB, which provides concurrent growth and access in a thread safe way. This is a replacement for a `vector`, whenever the deduction matches, e.g., when the programmer specifies a data collection which is *Random* and *Iterable*.

`cpriority_queue`: Concurrent priority queue wrapped from `concurrent_priority_-vector` of Intel oneTBB, which allows concurrent `push` and `pop` items with multiple threads where items are popped in a priority order as the sequential priority queue. This is a replacement for a sequential `priority_queue` whenever the deduction matches.

`cunordered_set`: Concurrent (unordered) set wrapped from `concurrent_unordered_-set` of Intel oneTBB, which maintains an unordered sequence of unique elements where concurrent *insertion*, *lookup*, and *traversal* are provided. This is a replacement for an `unordered_set` whenever the deduction matches.

`cunordered_map`: Concurrent (unordered) map wrapped from `concurrent_unordered_-map` of Intel oneTBB, similar to the just mentioned `concurrent_unordered_set`, it is an unordered associative container. It stores key-value pairs with unique keys where concurrent *insertion*, *lookup*, and *traversal* have been provided. This is a replacement for an `unordered_map` whenever the deduction matches.

`cset`: Concurrent set wrapped from `concurrent_set` of Intel oneTBB, which maintains a sorted sequence of unique elements where concurrent *insertion*, *lookup* and *traversal* are provided. This is a replacement for a `set` whenever the deduction matches.

`cmap`: Concurrent map wrapped from `concurrent_map` of Intel oneTBB, similar to the just mentioned `concurrent_set`, it is a sorted associative container. It stores key-value pairs with unique keys where concurrent *insertion*, *lookup*, and *traversal* are provided. This is a replacement for a `map` whenever the deduction matches.

`lockfree_stack`: A concurrent lock free stack wrapped from `lockfree_stack` of Boost's Lockfree, which maintains a LIFO sequence of data with a lock-free multi-producer/multi-consumer mechanism. This is a replacement for a `stack` whenever the deduction matches.

`lockfree_queue`: A concurrent lock free queue wrapped from `lockfree_queue` of Boost's Lockfree, which maintains a FIFO sequence of data with a lock-free multi-producer/multi-consumer mechanism. This is a replacement for a `queue` whenever the

deduction matches.

As we propose Collection Skeletons flexible property-based abstractions for data collections, more concrete data structures can be easily integrated following the same methodology, e.g., we can integrate containers from STAPL to Collection Skeletons as further candidates [130].

5.3.2 Thread Safety

When extending the prototype library of Collection Skeletons to incorporate Implicit Parallelism, we introduced concurrent data structures from Intel's oneTBB and the Boost Lockfree library. These data structures provide thread-safe operations, thus exposing further opportunities of parallelisation for Collection Skeletons.

A program is thread-safe if it functions correctly when executed simultaneously by multiple threads. This implies that no corrupted data structures, unpredictable behaviours, or other unexpected problems will result from concurrent thread activities. As discussed earlier, writing parallel code is challenging because thread safety must always be met to ensure correctness. When executing parallel programs, multiple threads access shared data or resources. Without a proper thread safety policy, issues such as race conditions, deadlocks, and data inconsistencies could arise. For example, thread-safety can be violated when two threads are trying to modify the same resource at the same time. Unlike flaws in a sequential program, a thread-unsafe concurrent program can compile and run without issues, but it might still produce incorrect results.

This work does not aim to address the thread safety challenges inherent in parallel computing. Ensuring consistently thread-safe programs across multiple platforms, while maintaining a simple programming interface, falls beyond our scope. Although the prototype has been extended with Intel's oneTBB and Boost's Lockfree library, our primary goal is to make it as straightforward as possible for programmers to write parallel code as a proof-of-concept. At the same time, we recognise that there might be exposure to thread-unsafe parallel functions because concurrent data structures from Intel's oneTBB are not fully thread-safe (there are certain thread-unsafe operations for some concurrent data structures). To facilitate smoother coding on the programmer's end, we offer compile time information about these thread-unsafe functions for our prototype library. This information can guide programmers, even if they are unaware that they are writing a multi-threaded parallel program.

Let us consider a simple use case for thread-unsafety —A programmer declares a

valid data collection with properties, which will ultimately be deducted as a `cunordered_set`, as shown in Figure 5.4. With Collection Skeletons, they have no idea about the concrete data structures which are hidden to them; then they invoke a member function `erase` as they want to delete an item from the data collection. However, the function `erase` associated to a `cunordered_set` is not thread-safe, which the programmers cannot know in advance. Consequently, programmers may inadvertently encounter risks associated with thread-unsafety. To help the programmer avoid potential nondeterminism behaviour, we introduce an alert system that provide information for this case which is after the program enters compilation, an alert information will be returned to the programmer suggesting the usage of a thread unsafe member function. However, this alert information will not interrupt the compilation, thus, the programmer would have two choices —1. Continue with this compilation while taking the risk that part of the program might be thread unsafe, which might or might not cause undefined behaviour; 2. Recompile the program after specifying a macro to make the multi-stage pattern matching algorithm not favour parallel for the generated code. As for the implementation of the alert system, a human-readable message will be printed through C++'s `static_assert`.

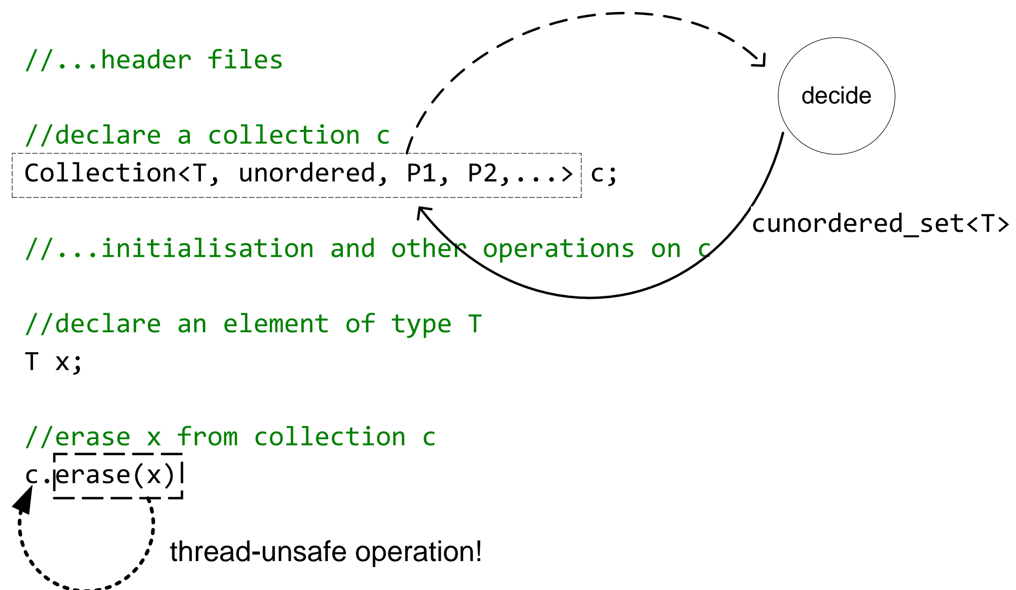


Figure 5.4: Example of thread unsafe with a concurrent data structure

It is worthwhile to consider whether thread-safety can be introduced as a non-functional property to Collection Skeletons. Similar to other non-functional properties like time complexity or space complexity, a programmer may also want to specify

whether a data collection guarantees thread-safety. With a thread-safe data collection, programmers can confidently focus on concurrent programming without worrying that thread-unsafe operations might impact the program's correctness. The introduction of thread-safety as a property will be the subject of our future research.

5.3.3 Beyond the Preparation for Parallelisation

As mentioned in Chapter 4, the central concept of this work is to present a novel programming model that allows programmers to write parallel and portable code with minimal efforts. As for a prototype for evaluation, the extended library is designed to generate code *as parallel as possible* —it firstly looks up the concurrent data structures to decide if a candidate can be founded; if there is no concurrent data structure as a feasible candidate, then it continues performing pattern matching on sequential data structures as discussed in Chapter 4.

With the generated concurrent data structures as concrete implementations, it is possible to go further beyond the concurrent data structures by inserting OpenMP's directives to parallelize operations associated with those data structures, e.g., inserting a parallel for directive before a for loop that operates on a concurrent data structure. Currently, for the purpose of prototype and evaluation, we manually locate and insert the OpenMP directives into the source code, which is shown in Figure 5.5

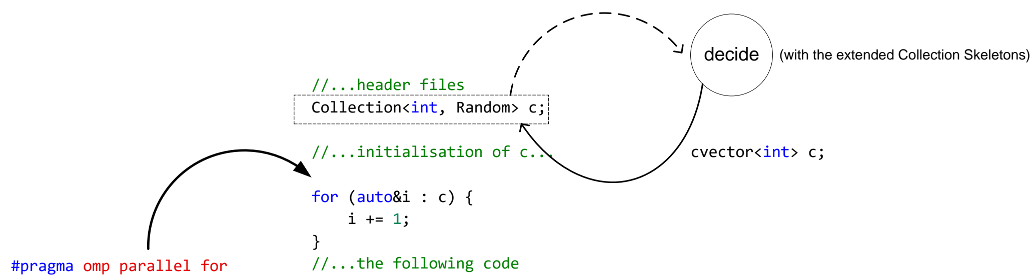


Figure 5.5: Inserting OpenMP directives into a context of concurrent concrete data structures

We focus on for loops that are associated with the property-based data structures to ensure there is no need to modify any logic structure in the program. In fact, OpenMP directives will only be inserted before for loops that include operations on the data collection. We emulate usage in which an auto-parallelizing compiler

would transparently insert OpenMP directives into the user code, potentially providing improved parallel performance beyond the concurrent data structures.

This process can be easily performed by a tool and be transparent to the programmer, e.g., with DawnCC inserting automatically OpenMP directives into the source code [100, 126]. We will evaluate the computational performance provided by such process in the following section.

5.4 Evaluation

We evaluated the computational performance of the extended prototype library using the same 17 benchmark programs with the same hardware & software configurations referenced in Section 4.6. We firstly tried to substitute the sequential implementations with concurrent implementations; as a result, access functions in the source code of the benchmarks that can be replaced with their parallel counterparts will also be replaced; as the implementation substitution happens at the level of the Collection Skeletons' concrete data structure candidates, therefore the source code of the rewritten benchmark programs remains unchanged. After that, we manually inserted OpenMP directives into the parallelised benchmark programs where there are for loops operations associated with the data collections. The insertion of OpenMP is to emulate a tool such as an automatic compiler performing further parallelisation beyond Implicit Parallelism. As before, a speedup greater than 1.0 indicates that the programs rewritten using Collection Skeletons exhibit a performance improvement compared to the sequential versions; conversely, values less than 1.0 indicates a decline in computational performance.

5.4.1 Evaluation on the Concurrent Data Structures

Not all of the 17 benchmark programs data collections lead to a concurrent data structure; in fact, during the manual rewriting performed in Section 4.6, we have identified that only some of these programs where a concurrent data structure introduced in Section 5.3.1 can substitute the corresponding original ones. For example, there is currently not a concurrent circular data structure supported, thus it is not possible to substitute the original data structure with a concurrent one. Table 5.1 displays the list of implicitly parallelised benchmark programs, highlighting those that have achieved performance boost using Implicit Parallelism. Additionally, through manual examination of the extended deduction rules provided by the modified multi-staged

pattern matching algorithm, we have identified the concrete data structures for these benchmark programs as shown in column *Details*. It should be noted that for these benchmarks no parallelism has been exploited, but the only change is that they now rely on a concurrent data structure for their implementation. Furthermore, SMT has been enabled for all three platforms, even though the concurrent data structures do not exploit multiple threads themselves. The threads information for these platforms can be found at Section 4.6. However, we have not manually configured the number of threads used on the three platforms at this stage.

Table 5.1: Benchmarks' Data Collection Substitution with Concurrent Data structures

Benchmark	Details	Speedup Intel Desktop	Speedup Arm Server	Speedup Intel Server
ising	concurrent_vector	1.06	0.99	1.00
set	concurrent_vector	1.62	1.52	1.02
tinn	concurrent_vector	0.95	1.01	1.00
mp3	concurrent_vector	0.94	0.65	0.82
scheduler	concurrent_priority_queue	0.29	0.33	0.19
md5	concurrent_unordered_map	0.68	< 0.01	0.42
infix	lockfree stack lockfree queue	< 0.01	< 0.01	< 0.01
kruskal	concurrent_unordered_map	0.70	0.90	0.66

Table 5.1 shows that 8 out of 17 benchmark programs are the cases where the data collection lead to concurrent data structures as implementations; among these 8 benchmarks, only set benchmark reports mild computational performance improvement across all platforms. Other benchmarks, however, report various performance decrease across the platforms. However, some cases benefit from concurrent data structures even when executed sequentially —this might be because the concurrent data structures are better implementations for the benchmark programs, or they can initiate better optimisation for the compiler, which is subject to our future research. In conclusion, a major number of the 8 benchmarks report performance decrease while only one benchmark report performance increase with a substantial speedup of up to 1.62.

The experimental results are still acceptable given we have only done minimum adaptations on our prototype library without touching the benchmark programs; besides, concurrent data structures tend to introduce more overhead compared to sequential data structures. Furthermore, when looking at the source code of the 8 modified benchmark programs, we have observed extremely few opportunities for parallelisation through parallel access functions, as, for example, there are few access functions in the source code that can be substituted with a parallel one. In fact, even if we had

implemented parallel access functions in this prototype, the computational performance by the Implicit Parallelism, where in combination of the concurrent data structures and the parallel access functions, could still be limited. Nevertheless, it is reasonable that implicit parallelism can be more profitable where there is intensive usage of access functions such as `sort` and `find` for which parallel implementations can be provided as part of the Collection Skeletons library.

5.4.2 Evaluation with the Introduction of OpenMP

The simple introduction of concurrent data structures seem to be not beneficial according to the computational performance. However, there is parallelism in the 8 benchmarks mentioned in Section 5.4.1, but not at the level of access functions. Instead, this parallelism is at the loop level and contained in the user code, which would need to be modified to exploit this parallelism. However, modifying the user level application code is not “implicit” —the purpose here is to show that there is parallelism to be exploited in combination with the concurrent data structures, but which might require the user to rewrite the application. We manually inserted OpenMP directives into these benchmarks where there are for loops associated with the data collections. No other code rewriting or optimisation have been introduced as we aim to emulate a tool that provides parallelization beyond the Implicit Parallelism. Similarly, we have kept the SMT enabled for all platforms to ensure OpenMP is able to fully utilise the threads on each platform.

Table 5.2 presents the speedup of these 8 benchmarks after being extended with OpenMP directives. This table also includes the number of threads used by OpenMP on three platforms, where 12 threads have been used on Intel Desktop, 4 threads have been used on Arm Server, and 72 threads have been used on Intel Server. We configured the number of threads for each platform according to the system configuration introduced in Section 4.6, where we deliberately set the number of used threads equal to the number of hyperthreads (cores for Arm Server). Among the 8 benchmarks, set, ising and tinn benchmarks report computational performance improvement across all platforms; while performance decrease is reported by benchmark md5 and infix compared to the sequential ones. Benchmark mp3, scheduler and kruskal report performance boost on Intel Desktop and Intel Server while decrease on the Arm Server. In conclusion, a major number of cases report performance increase with a speedup up to 5.64 for the best case, which is due to 72 threads together contributing to the computation. Compared to the

evaluation in Section 5.4.1, the computational performance with OpenMP introduced is much better.

Table 5.2: Parallelisation of the benchmarks with OpenMP

Benchmark	Details	Speedup Intel Desktop number of threads=12	Speedup Arm Server number of threads=4	Speedup Intel Server number of threads=72
ising	concurrent_vector	1.25	2.09	5.64
set	concurrent_vector	1.80	1.09	5.29
tinn	concurrent_vector	2.49	1.71	1.01
mp3	concurrent_vector	2.27	0.60	4.53
scheduler	concurrent_priority_queue	1.29	0.97	1.67
md5	concurrent_unordered_map	0.68	< 0.01	0.42
infix	lockfree stack lockfree queue	< 0.01	< 0.01	< 0.01
kruskal	concurrent_unordered_map	1.40	0.90	1.31

These performance boosts are achieved from OpenMP's multi-threading parallel performed upon the concurrent data structures from Intel oneTBB as concrete implementations. Comparing to the sequential cases where only a single thread is used to execute the whole program, OpenMP makes the program be executed by multiple threads and thus exploits more parallelism out of these benchmark programs, while the concurrent data structures ensure the thread safety of the simultaneous operations. Three benchmark programs only report performance boost on specific platforms, e.g., mp3 benchmark program reports substantial performance boost on Intel Desktop and Intel Server with a speedup of 2.27 and 4.53, separately; however, there is performance down reported from the Arm server —it could be the different architecture, or that the I/O capacity of the Arm server is weak as mp3 benchmark is an I/O intensive program. For benchmark program ising and set, there is a tendency that Intel Server reports much better performance increase than the other two platforms; this could be because compared to the other two platforms, the Intel Server fully used as many as 72 threads that execute on the operations related to the concurrent data structures, suggesting great scalability of parallelism for these benchmark programs. The benchmark infix reports substantial performance decrease on all the tested platforms, which could be due to the strict access order in this program that hinders the parallelisation of the associated operations. Furthermore, the different results from different platforms suggest the possibility of selecting the optimal data structure tailored to a specific platform, as discussed in Chapter 4, which can be implemented by swapping out for optimal data structures during compile time or runtime targeting for a more optimal platform.

The experimental results are encouraging, given that we merely integrate concurrent

data structures as the concrete data structures while inserting basic OpenMP's directives into the source code, which can be easily automatically with the help of tools such as [126]. This can be another future research direction, as we integrate more concurrent data structures as concrete implementations, it is reasonable to expect that more parallelism can be exploited in various ways based on these data structures.

Furthermore, the evaluation results for combination of concurrent data structures and OpenMP directives open up further parallelisation opportunities —it is possible to let the users specify parallelised algorithmic skeletons instead of writing for loops and waiting for a tool to make it parallel. We will discuss this in the next chapter.

5.5 Chapter Summary

In this chapter, we defined **Implicit Parallelism** and discussed the extension of the Collection Skeletons to promote parallelisation through implicit parallelism by integrating parallel-friendly or concurrent data structures as implementation options together with parallel access functions. Ideally, with implicit parallelism enabled, the programmers can specify data collection with properties and the Collection Skeletons generate concurrent data structures and their associated parallelised access functions when appropriate.

We further extended the prototype library with concurrent thread safe data structures as a proof-of-concept and repeated the evaluation as introduced in Chapter 4 to measure the performance. The evaluation results suggest that the introduction of concurrent data structures can only offer limited computational performance benefit but mostly negative performance impact. However, Implicit Parallelism can be more profitable for specific problems, e.g., those have intensive use of access functions such as find and sort, where parallel access functions can better exploit parallelism. To explore further parallelism beyond implicit parallelism, we introduced OpenMP directives inserted into the source code and showed that the computational performance has greatly improved. Therefore, providing thread-safe implementations is a proper preparation to further exploit parallelism. As the lack of internal parallelised data structures as concrete implementations, we were not able to investigate the performance variation after introducing internal parallelised data structures.

Recognising the inherent limitations of implicit parallelism in terms of enhancing computational performance, as well as the parallelism beyond implicit parallelism which can lead to better parallel performance, we seek to further capitalise on opportunities to incorporate parallelisation and its performance benefits. Therefore, in the next chapter,

we will delve into explicit parallelisation, integrating it with algorithmic skeletons to make the users explicitly specify abstractions for algorithms in cooperation with the property-based data abstractions.

Chapter 6

Explicit Parallelism —Integration of Collection & Algorithmic Skeletons

In this chapter, we delve into **Explicit Parallelism** and its implementation, focusing on the expansion of the Collection Skeletons through integration with Algorithmic Skeletons. As introduced in Chapter 4, Explicit Parallelism defines parallelism that is exposed to the programmers whose active involvement is required, which is in contrast to the Implicit Parallelism introduced in Chapter 5. Our goal is to not only make the algorithms from these algorithmic skeletons available to the Collections Skeletons, but also to tailor both skeletons according to the properties of the data collections. We discuss the interactions of our Collection Skeletons with Data-centric algorithmic skeletons, e.g., `map`, as well as prevalent algorithmic skeletons such as stencil. We extend the prototype library to integrate with multi-threading parallel algorithmic skeletons, and evaluate the computational performance of the expanded prototype library for Explicit Parallelism, which mirrors the evaluations conducted in Chapter 4 and Chapter 5.

6.1 Introduction

Having introduced the Collection Skeletons and their extension for Implicit Parallelism, programmers can now develop portable and parallel programs using property-based abstractions. Nevertheless, a gap persists between the current Collection Skeletons and the broader scope of parallel computing. Regarding the motivation example we highlighted in Section 1.2.3, the existing Collection Skeletons struggle to parallelize this instance. Even with our extended prototype library, which incorporates concurrent data structures, parallelizing the code segment inside the *for loop* from the motivation

example that continues to be challenging. In essence, relying solely on collections and their member functions yields limited capability for parallel computing.

Let us review the structure of a computer program and explore the process of developing programs to gain a fundamental understanding of parallel computing. In a computer program, the algorithm and the data structure are the most critical components [136]. External input data must be properly stored within a data structure to allow for effective manipulation by the corresponding algorithm. This insight is encouraging, given that we have already proposed a robust solution for the data structure/container aspect. It would be advantageous to effect similar enhancements on the algorithmic side to ensure that both aspects operate in harmony.

In many parallel computing frameworks and toolkits, the *for loop* is a primary focus when considering the parallelisation of an existing sequential program or developing parallel code. For instance, OpenMP offers a *parallel for* directive as a foundational tool to parallelize a for loop. Similarly, the Parallel Patterns Library and Taskflow furnish a `parallel_for` clause to construct a for loop that executes in parallel.

Simply parallelizing *for loops* or writing parallel iterations is not sufficient to achieve practical and efficient parallel computing. There are instances in which we must create functions to execute in parallel, or we may wish to parallelize a sequence of code with existing dependencies (as previously discussed). To address these needs, parallel computing paradigms, such as task-based parallelisation and the consumer-producer model, have been introduced. Leveraging these advanced parallel computing toolkits allows us to develop parallel programs that can encompass a wider range of their sequential counterparts. This, consequently, provides greater opportunities to exploit performance enhancements through multi-threading. However, increasing parallelism concurrently escalates the complexity of programming. This escalation confronts programmers with the fundamental problem highlighted in this thesis: the challenge of writing parallel code.

To facilitate the task of writing parallel code while preserving extensive parallelism, an abstraction layer for parallel computing is indispensable. Such an abstraction ought to conceal the complexities of parallelism from the user, while providing a user-friendly and robust set of programming APIs. This requirement dovetails with the discussions presented earlier in this thesis on Collection Skeletons. As demonstrated in preceding chapters, Collection Skeletons permit programmers to bypass the overspecification associated with data collection, thereby enabling the development of more flexible and portable programs. Consequently, it is beneficial to augment Collection Skeletons with

parallel operation abstractions.

As introduced in Chapter 1, Algorithmic Skeletons constitute a high-level parallel programming model that offers an abstraction over a set of algorithms for parallel and distributed computing. The `map`, `reduce`, and `filter` functions are well-known algorithmic skeletons that have been widely implemented across various programming languages and toolkits. Given that Collection Skeletons have been inspired by Algorithmic Skeletons, it is imperative to merge the strengths of both skeletons to form a more potent programming model.This union should continue to enable programmers to write code that is both portable and adaptable, maintaining the simplicity introduced in previous chapters. Unlike the Implicit Parallelism introduced in Chapter 5, which hides parallelism from programmers, the integration of algorithmic skeletons with Collection Skeletons requires programmers to explicitly invoke the algorithms. The underlying implementation details are then managed by the library. Therefore, we term this integration of algorithmic skeletons with Collection Skeletons as **Explicit Parallelism**.

Definition 2. ***Explicit Parallelism of Collection Skeletons —Explicit Parallelism*** *specifies parallelism that is explicit to the users of Collection Skeletons, where the* ***users*** *need to* ***explicitly*** *use parallelisation means provided by Collection Skeletons, e.g., parallel algorithmic skeletons, to develop parallel programs.*

By integrating Explicit Parallelism into Collection Skeletons, we can further optimise the use of concurrent concrete data structures, as outlined in Chapter 5. When parallelised algorithmic skeletons operate on thread-safe data collections, they collectively boost computational performance while simplifying parallel programming for developers.

6.2 Integration of Both Types of Skeletons

As previously noted, the integration of algorithmic skeletons with Collection Skeletons is not a mere ad-hoc process. It is not simply a matter of introducing algorithmic skeletons and indiscriminately applying them to containers, a practice that overlooks potential compatibility issues, as observed in frameworks like SkePU. Instead, algorithmic skeletons should be carefully coordinated with Collection Skeletons, conforming to property-based abstractions.

6.2.1 Challenges on Exposing Properties for Algorithmic Skeletons

It appears plausible to adopt the methodology previously applied to the Collection Skeletons for the algorithmic skeletons. This would empower programmers to specify an algorithmic skeleton based on desired properties without delving into the implementation details. However, two questions emerge: 1. Can the declarative approach be effectively replicated for algorithmic skeletons? 2. If the answer to the first question is "yes", how can the algorithmic skeletons be integrated seamlessly with the property-based abstractions of the Collection Skeletons?

Let us examine a basic yet prevalent algorithmic skeleton, the `map`. Approaching from the reverse perspective, which properties would be essential to define a `map` skeleton? By definition, the `map` function is a higher-order function that applies a given function to each element in a collection. Semantically, the `map` skeleton comprises two principal components: the function to be applied and the collection containing the data to which this function is applied. It is important to note, however, that the function and the collection themselves do not constitute properties that define an algorithmic skeleton. Furthermore, although the sequence of function applications in a `map` may conform to a "happens-before" relationship, the order of execution is not a defining criterion for determining whether a set of computations qualify as a `map`. Similar reasoning can be applied to other algorithmic skeletons, as they typically prescribe a series of computations to be applied to a collection. Figure 6.1 gives an example of trying to identify the properties of a `map` skeleton, where `P1` and `P2` are rules of the algorithmic skeleton. Unlike extracting properties from data collections, this methodology is not practical.

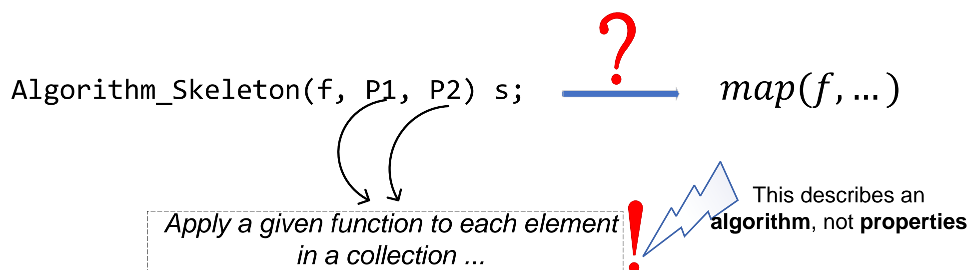


Figure 6.1: Identifying *properties* for a map skeleton

Consequently, establishing a strict property-based abstraction exclusively for algorithmic skeletons is not feasible. However, considering that these skeletons are ultimately applied to collections, and given our current ability to define data collections based on desired properties, it is pertinent to assess the interrelationship between the properties of collections and those of algorithmic skeletons. Specifically, it is necessary to identify the properties that a collection must exhibit to enable the effective application of a specified algorithmic skeleton.

6.2.2 Properties of Collections That Relate to the Application of an Algorithmic Skeleton

Let us examine the `map` skeleton further, using it as a case in point. As described in Figure 2.4 in Chapter 2, a `map` applies a function f to every individual element of a collection. To ensure that function f is applied to each element within the collection, what properties must the collection exhibit? In this context, it becomes evident that the collection must, at a minimum, be *Iterable* —it is imperative to be able to traverse each element in the collection to apply the function f uniformly. Given that the *Iterable* attribute represents the foundational requisite, we categorise *Iterable* as the **Necessary** property for a collection intended for interfacing with the `map` skeleton. Figure 6.2 presents the necessary properties, *Iterable*, of a `map` skeleton, where functions such as `iter()` or `next()` are required to facility a `map` operation over a collection.

Nevertheless, the mere property of being *Iterable* is insufficient to facilitate a *parallel* `map` wherein the function f can be executed concurrently. As is also shown in Figure 6.2, to achieve parallelisation of the function f across the collection's elements, the collection must additionally support *Random*. This provision ensures simultaneous access to all elements, thereby facilitating concurrent operation. It is worthwhile to note that the *Random* property is distinct from the *Iterable* property, which merely facilitates elemental traversal. Consequently, the **Sufficient** property enabling a parallel `map` operation —through the concurrent application of function$f$ to the collection's elements —is *Random*. This property not only encompasses *Iterable*, but also introduces the requisite functionality for parallel access.

Definition 3. ***Necessary Properties —*** *The minimum set of properties that facilitate an algorithmic skeleton's operation over a data collection.*

Definition 4. ***Sufficient Properties —*** *Extended set of properties that facilitate parallel operation for an algorithmic skeleton over a data collection.*

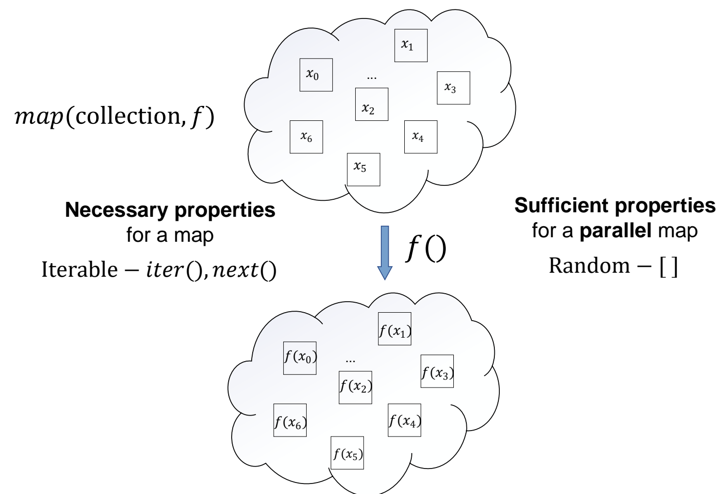


Figure 6.2: Necessary & Sufficient properties of a (parallel) map

However, it is not possible to decide whether a data collection can be applied to an algorithmic skeleton solely based on the properties of the data collection. In fact, for example, the properties of the function to be applied by a `map` skeleton also influence whether it can be applied in parallel.

For example, sometimes it is not possible to perform a `reduce` skeleton in parallel even if the data collection has the necessary and sufficient properties; in this case, they are *Iterable* and *Random*, which is to be discussed in Section 6.3.

```
//definition of a collection c which is of random accessible
Collection<int, Rnd> c;
...
//definition of a binary function f
int f(int x, int y){
    return x/y;
}
//reduce collection c with function f
auto b = reduce(f, c);
```

Listing 6.1: A reduce with a non-associative function

In Listing 6.1, the properties of collection `c` meet the requirements for a parallel "reduce" as described in Section 6.3. However, this `reduce` operation is incorrect because division is neither associative nor commutative.

It is important to note that these properties from algorithmic skeletons can be difficult to check, for example, it is nearly impossible to decide whether a function

is commutative when the function is quite complex. In this thesis, we do not discuss the properties from such input functions from the algorithmic skeletons. We focus on the necessary properties and sufficient properties of the data collections based on Chapter 4 and leave those properties from input functions of algorithmic skeletons for future research.

Based on the definition of **Necessary properties** and **Sufficient properties**, we analyse and discuss more details on their interaction with algorithmic skeletons and Collection Skeletons in the following sections.

6.2.3 Property Check for Collection Skeletons with Algorithmic Skeletons

Having proposed the concepts of **Necessary** and **Sufficient** properties to facilitate the application of specific algorithmic skeletons to data collections, it is feasible to perform a check prior to execution. Similar to how raising compilation details for an incompatible set of properties, as described in Chapter 4, prevents indeterministic behaviours and assists programmers in correcting their code, checking properties for "Collection Skeletons" when integrating specific algorithmic skeletons can provide analogous benefits.

Therefore, we introduce a property check process for each algorithmic skeleton that is integrated with the Collection Skeletons, which is shown in Algorithm 2. Every time an algorithmic skeleton is to be applied on a data collection, both the **Necessary** and **Sufficient** properties of the algorithmic skeleton are checked against the given collection. Algorithm 2 presents the procedure for the property checking.

Similar to the procedure for determining concrete data structures in Algorithm 1, Algorithm 2 adheres to a two-phase pattern matching mechanism. Initially, it assesses the **Necessary** properties against the data collection, as is shown in line 1, where the set of necessary properties of an algorithmic skeleton `Algo` is checked against the set of properties of the collection. If the collection possesses the necessary properties for the algorithmic skeleton, and this initial test is successful, the algorithm then further evaluates the **Sufficient** properties of the algorithmic skeletons in relation to the **properties** of the data collection, as shown in line 2. However, if the comparison against the **Necessary** properties is unsuccessful, the algorithm returns with none, meaning the necessary (nor Sufficient) properties of `Algo` are not satisfied. If the first phase is successful, but the second is not, the algorithm returns only *Necessary*, suggesting

Algorithm 2: Necessary & Sufficient Property Check

Input : Properties $\mathscr{P}=P_1,P_2...P_n$,
Algorithmic skeleton $Algo$

Output : Necessary/Sufficient properties satisfied or not

```
1 if (Necessary(Algo) ⊆ 𝒫) then
2     if (Sufficient(Algo) ⊆ 𝒫) then
3         return Necessary and Sufficient
4     else
5         return Necessary
6 else
7     return none
```

only necessary properties are satisfied thus sequential algorithmic skeleton over the data collection is feasible. Upon successful completion of both phases, the algorithm returns *Necessary* and *Sufficient* and the algorithmic skeleton can be applied to the data collection in parallel.

Based on Algorithm 2, we develop a property-checking mechanism for algorithmic skeletons with C++ metaprogramming as a further extension to our prototype library. Given that the concrete data structure and the algorithmic skeletons to be applied can be both decided at compile time, we introduce a compile-time property checking after the deduction of the concrete data structure.

Following the implementation of the prototype library introduced earlier, the example implementation is showcased in the following code listings as Figure 6.3, if the properties of the collection `c` match the *Necessary* properties and *Sufficient* properties for a parallel `map`, a parallelised version of a `map` skeleton will be generated; or if the properties of `c` only match the necessary properties of a `map`, a sequential version of a `map` skeleton will be generated; otherwise, the compilation will be interrupted and human-readable information will be returned as compilation error.

6.2.4 Property Inference

While it is not possible to infer the complete set of properties of the resulting collection after application of an algorithmic skeleton, we can at least infer *some* of the properties of the collection resulting from the application of a *map* on another collection. For example, we know that the resulting collection is of smaller or the same size as the

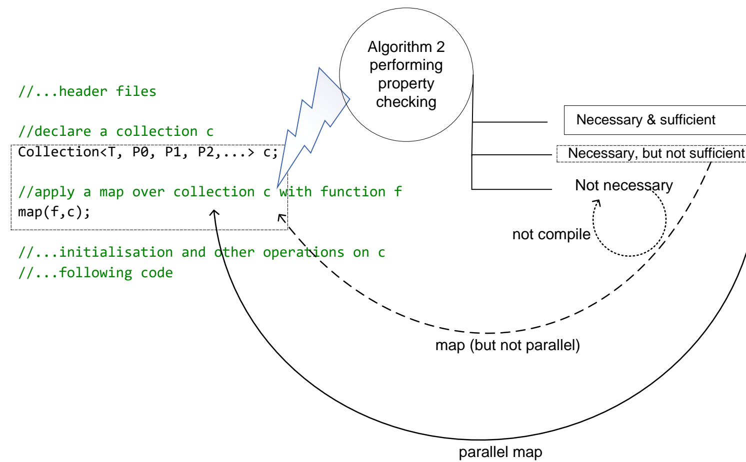


Figure 6.3: Property checking for a map skeleton over a data collection

original collection. As the function f maps each element of the original collection to exactly one value, the size of the resulting collection is limited by the size of the original collection. However, as duplicate values might be introduced by f, a collection that does not allow duplicates, e.g., a set, the resulting collection might end up with fewer elements.

Typically, we cannot make any statement on the order of the resulting collection or uniqueness of elements under a *map* without further information about the function f. For this reason, we require the user to specify the desired properties of the resulting collection, thereby following our declarative paradigm where the user is asked to state the properties they rely on. As mentioned earlier in Chapter 4, the programmer is responsible for the return value's type, as the Collection Skeletons are designed to be flexible.

As a result, we do not implement a property inference feature in our prototype library even after the extension with algorithmic skeletons —the user needs to explicitly specify the return collection with properties. However, as the implementation of algorithmic skeletons with C++ as in our prototype library, we need to define the type of the return value in corresponding to the input data collection, and the programmer can not modify the type of the return value once the input data collection and the algorithmic skeletons have been presented. For interacting with the extended prototype library, the programmer is strongly advised to use `auto` keyword, which is a type inference feature introduced by modern C++. To achieve this, we follow the Scala standard library to implement the return type of the algorithmic skeletons. We will detail the implementation in the prototype library in the subsequent sections.

Even though property check and property inference are applicable, it is still the responsibility of the programmer to provide collections to the compute skeletons that have the correct properties. Therefore, the programmers need to particularly understand the interface functions of the algorithmic skeletons that they intend to apply to the collection to ensure both skeletons are consistent.

6.3 Data-centric Algorithmic Skeletons

As briefly introduced early, `map`, `reduce` and `filter` are popular algorithmic skeletons that are widely applied in several areas. They are also included in standard libraries of several programming languages such as Scala and Java. As the name suggested, data-centric skeletons focus on the data that are stored in the data collection —the operation tends to be functional and without side effects. These algorithmic skeletons are straightforward to apply in practical programming and there have already been numerous implementations, sequential and parallel, for these skeletons.

6.3.1 Map

As introduced earlier, a `map` skeleton applies a function to each element of a collection. A `map` is a data-centric algorithmic skeleton as it interacts with the elements of the collection with the function to be *mapped*. As long as the collection is Iterable, the collection can be applied with the function as a `map` skeleton, which is the necessary property discussed early. However, a collection can be mapped in parallel only if an iterable collection can also be accessed randomly. Below are the set of *Necessary* properties for a `map` skeleton and the set of `Sufficient` properties for a parallel map.

Necessary properties- Iterable

Sufficient properties- Random

With the specified *Necessary* and *Sufficient* properties, the programmer can more effectively harness explicit parallelism by precisely invoking algorithmic skeletons to achieve optimised performance.

Though currently the extended Collection Skeletons do not try to infer the return collection's property of an algorithmic skeleton, there does exist the opportunity to infer the type of the return value of a `map` skeleton. Furthermore, we would like to know more about the return value of each algorithmic skeleton introduced as the extension, thus making the programmer more confident to work with them. Based on the semantics

of the `map` skeleton and Figure 6.2, it is not difficult to consider the return type should be the same as the input data collection's type. For example, if we `map` a vector of 4 integers with a function "add by 1", then the return data collection should still be a vector as the function only applies to the elements inside the vector, not the vector itself, which is shown in Figure 6.4. Although the programmer can freely modify the `map` to make it return a list instead, however, we do not discuss this case for our extended Collection Skeletons.

$map(f, \text{collection})$, where $f(x): x \to x+1$

1	2	3	5

A vector of size 4

2	3	4	6

A vector of size 4

Figure 6.4: A map over a vector

However, the type persist does not necessarily mean all the properties and parameters of the data collection shall persist. Let us consider the example in Figure 6.5.

```
//declare a collection c
Collection<int, Iterable, Associative, Unique> c = {-1,1,2,3};

//definition of function f
int f(int x){
    return abs(x)
}

//map function f over c
map(f, c);
                {-1,1,2,3} -> {1,2,3}
```

decide

E.g., std::set<int>

Different size!

Figure 6.5: Example of mapping a collection

The programmer has declared a data collection with properties *Unique*, *Iterable* and *Associative* to store 4 unique integers of $\{-1, 1, 2, 3\}$, which can be a sequence or a set. Then, they would like to `map` this data collection with a function `abs` that returns the absolute value of the input element. If the operation is applied over a *set*, however, upon applying the function (let us assume before the merging happens in the *set*), the results should be $1, 1, 2, 3$, which is a sequence but conflicts with the uniqueness of a *set* if a set will be the return data collection. Given the semantics of a set, particularly

the property of *Uniqueness*, duplicated elements are not allowed, and the result should be $1,2,3$ if a set is to be returned. However, the size of the returned data collection does not correspond to that of the input, even though the properties remain unchanged and the type of data returned is the same. The size discrepancy between the input and returned data collections can only be ascertained at runtime, regardless of the prototype's implementation. This characteristic renders it challenging, if not impossible, to determine the size variance prior to execution, which we have also discussed in Chapter 1. In practical programming, it is also possible for other parameters of the returned data collection to differ from those of the input.

6.3.2 Reduce

The *reduce* skeleton is a common algorithmic pattern frequently used in conjunction with the map skeleton. *MapReduce* is a combined pattern that has been used in several areas, and that has been extended towards parallel/big data frameworks such as Hadoop. Different from a `map`, a `reduce` applies to each element of a collection with a function f, where f operates among these elements.

Similarly, the *Necessary* properties for a `reduce` on a collection should be *Iterable*; Moreover, the *Sufficient* property is *Random*. At this stage, we disregard the commutativity of the function f, which could impact the determinism of a parallel reduce.

Necessary properties- Iterable

Sufficient properties- Random

After applying a `reduce` operation to a data collection, the return value should be of the same type as the elementary type of the data collection `T` and have a size of 1. Thus, for a `reduce` operation, whether performed in parallel or not, the return value is determined.

6.3.3 Filter

The `filter` is another algorithmic skeleton that often works in conjunction with `map` and `reduce`. Filter works exactly as a filter —Given a data collection and a Boolean function f, the Boolean function f checks each of the elements of the data collection; whenever the result is false, the checked element will be dropped, and the remaining elements will be returned together.

Similar to `map` and `reduce`, *Iterable* is the *Necessary* property for a `filter`, and *Random* is the *Sufficient* property for this algorithmic skeleton to enable parallel operation.

Necessary properties- Iterable

Sufficient properties- Random

Different from a `map` skeleton, a `filter` against a collection should return a collection no larger than the input one. While we can determine the return type, we cannot know the size of the resulting data collection before its return as the nature of this skeleton. While programmers can implement their own versions of `filter` in practical programming, allowing them to return data collections of different types with varying properties, we maintain the convention of making the return type of a `filter` skeleton the same as the input data collection.

6.3.4 Zip

The `zip` skeleton is a common function, especially in functional programming languages such as Haskell and lisp. A `zip` skeleton merges two data collections, and the result is a data collection of pairs of tuple elements from both collections. The code and figure below presents an example of a `zip` skeleton applied to two data collections of the same size.

Similar to a `map` skeleton, *Iterable* is the *Necessary* property for a `zip` skeleton, while *Random* is the *Sufficient* property for a parallel `zip`.

Necessary properties- Iterable

Sufficient properties- Random

When merging two input data collections, the resulting data collection can be challenging to determine. This is particularly true when the two input data collections have different properties or types, making it tricky to establish the return value. Manually defining the result of data collection can resolve this uncertainty. For example, the return data collection type can be made consistent with either the first or the second input data collection. Therefore, the properties of the return data collection can also be manually defined in this case. Furthermore, the programmer can even define a data collection that is entirely distinct from the two input collections. In the extended prototype, we manually specify the return type of the data collection to ensure greater consistency with other parts of the library.

6.3.5 Implementation of the Data-centric Algorithmic Skeletons

As a proof-of-concept, we have implemented algorithmic skeletons based on several existing libraries, including Taskflow [70], oneAPI [29], and FunctionalPlus [47] to adapt to our prototype library. Similar to the integration of concrete data structures from STL, Boost, and other third-party libraries, where a wrapping technique has been applied, we continue to wrap some algorithmic skeletons from different libraries.

Most of the C++-implemented algorithmic skeletons expose parameters as iterators, which is consistent with C++'s standard. While this approach may offer good performance and adhere to standard C++ programming practices, iterators-based programming APIs are not compatible with the extended prototype of our Collection Skeletons. We aim to avoid exposing excessive implementation details to programmers, including iterators in the function interface. Although the *Iterable* property introduces functions related to iterators, these functions primarily involve calculations based on their positions, and programmers typically do not manipulate the iterators directly, which in C++ are pointers underneath. For a convenient yet concise prototype for the extended Collection Skeletons, we aim to enable users to work with both skeletons seamlessly.

```
map(function, collection)
```

In this scenario, the programmers select an algorithmic skeleton, e.g., a `map` to be applied to a collection along with a specific function. This usage is highly abstract and concise, making the computational process more transparent to programmers. Programmers only need to concern themselves with defining the properties their data collection should have and selecting the algorithmic skeleton to apply to the collection. Implementation details for both data collections and algorithmic skeletons are hidden from them.

Table 6.1: Programming API of selective algorithmic skeletons

Algorithmic Skeletons	Programming API
map	`map(function, collection)`
reduce	`reduce(function, collection)`
filter	`filter(function, collection)`
zip	`zip(collection1, collection2)`

Table 6.1 lists the programming APIs of the algorithmic skeletons introduced earlier. All of these data-centric algorithmic skeletons in this extended prototype provide both sequential and multi-threaded parallel versions.

With the extended prototype library, programmers can write parallel code that runs multi-thread with minimal effort.

For example, if a programmer has a set of integers and wants to store these data into a variable data collection, anticipating that more data might arrive later, there is no problem. In this case, they can simply specify the property as *Variable*. Furthermore, the data collection should also be *Iterable* since they want to perform some operations afterwards. Additionally, the *Random* (denoted as *Rnd* in the following code listing) property is also required because they would like to access certain elements based on their index. Therefore, the data collection can be declared as

```
//declare collection c and initialisation
Collection<int, Variable, Iterable, Rnd> c {...};
```

The programming task is that they want to square all the elements. They can easily write a for loop in C++ programming. However, if they want to parallelize the for loop, they would have to delve into other parallel frameworks, which can be tedious. However, they realise that the operation is essentially a `map`. Since the prototype library has been extended with algorithmic skeletons, all they need to do is to specify the function as a parameter for the `map` operation as,

```
//definition of a function f
int f(int x){
    return x*x;
}
//map collection c with function f
    map(f, c);
```

But now the question arises: How to make it compute in parallel? This time, with Collection Skeletons, they do not need to do anything. As long as the collection can be mapped in parallel, which is the case in this example, the `map` operation can be executed in multiple threads. Thanks to the extended Collection Skeletons, programmers can write neat parallel code without needing to understand the computational details. As is shown in Figure 6.6, a parallel `map` is eligible with the specified properties.

However, if they changed their mind initially, e.g., they do not want the property *Random*,

```
//declare collection c and initialisation
Collection<int, Variable, Iterable> c {...};
```

Next, they want to perform the same operations using the same `map` skeleton

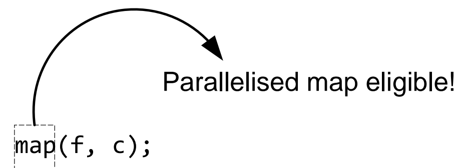


Figure 6.6: A parallel map is eligible given the properties of the collection c

```
//map collection c with function f
    map(f, c);
```

This is a legitimate approach and will return the correct value in the end. However, this time there will be no parallel code for this map; instead, it will result in a sequential code, essentially a sequential map skeleton, which is shown in Figure 6.7.

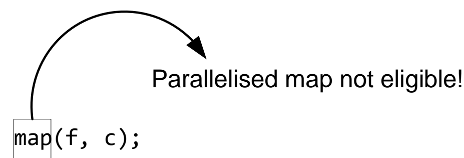


Figure 6.7: A parallel map is not eligible given the properties of the collection c

Since the implementation details have been hidden, the developers of the prototype library have more opportunities to optimise both types of skeletons and, consequently, the entire program. Regarding the implementations of the data-centric algorithmic skeletons, different versions of multi-threaded algorithmic skeletons can be provided, allowing for the swapping out of algorithmic skeletons at compile time, just like the concrete data structures swapping out as introduced in Chapter 4. Similar to the prototype Collection Skeletons where we currently only support limited compile time adaptation on data structures; we have yet to implement more and smarter swapping out for algorithmic skeletons, which is a subject of our future work.

It can be concluded that for all the data-centric algorithmic skeletons discussed in this section, the sufficient property for a parallel computation is *Random*. When programmers want to express parallelism for a data-centric algorithmic skeleton, they should specify a collection with the *Random* property.

6.4 Advanced Algorithmic Skeletons

Data-centric algorithmic skeletons are relatively straightforward to implement and integrate into Collection Skeletons because the focus is primarily on storing data in the collection, involving minimal logic computations and control flows. However, data-centric algorithmic skeletons have limitations when it comes to broader applications as they only cover a limited range of scenarios, e.g., converting a small number of loops into a series of data-centric algorithmic skeletons.

To enable programmers to write parallel code for a broader range of problem domains, diverse algorithmic skeletons that support complex computations are necessary. Therefore, we introduce three algorithmic skeletons: stencil, wavefront, and divide-and-conquer. We have chosen these three algorithmic skeletons because they are either widely used in many scenarios or closely related to the properties of collections introduced in Chapter 4.

6.4.1 Stencil

6.4.1.1 Introduction to Stencil Skeletons

The Stencil skeleton has found wide use in various domains, including data analysis and image processing. With the proliferation of computer hardware, especially GPUs, the Stencil skeleton has been adapted to multiple platforms to harness parallelism. Stencils come in various forms, ranging from one-dimensional to multi-dimensional, and from one-point to several-point neighbour references. A simple yet typical stencil skeleton is presented in Figure 6.8.

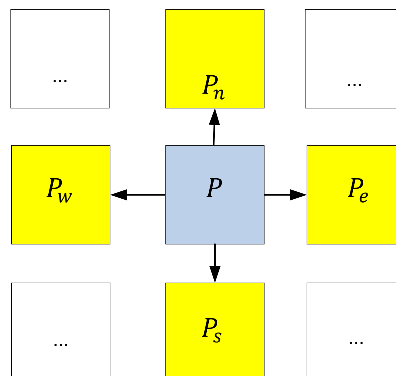


Figure 6.8: A simple stencil over a matrix

Where the input data are organised as a two-dimensional vector (matrix). For an element P from the matrix, the computation for a new P will be operated by a function over its four neighbours from east, south, west and north, where they are denoted as P_e, P_s, P_w, and P_n, separately. Such computation will be performed for every element of the matrix. When an element has less than four neighbours, e.g., at the corner of the matrix, a *border policy* will be applied to compute the value for that element, which is shown in Figure 6.9. It is generally difficult to interpret stencil skeletons with more complex structures as figures, therefore, we focus on a set of stencils that can be represented with a collection and a stencil rule (for the instance mentioned, the stencil rule is to calculate the four directions of the element's neighbourhood).

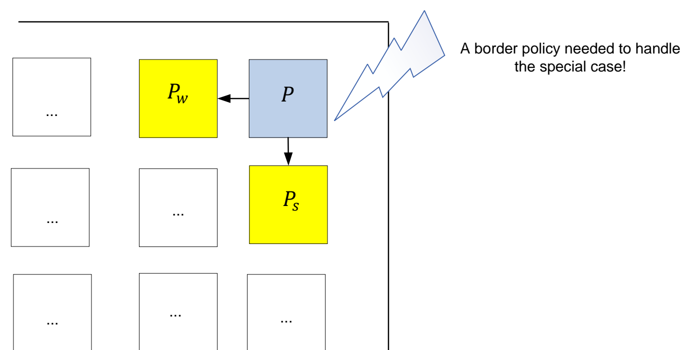


Figure 6.9: A simple stencil over a matrix —the border case

As stencils cover a set of parallel computing pattern and are widely used in multiple areas, extending the Collection Skeletons will benefit both skeletons as a more powerful concept of programming model that works at an abstract level.

Neighbourhood A collection has the *Neighbourhood* property if each element of the collection has neighbours, i.e., where elements have a specific spatial or logical relationship to each other; otherwise, the collection is not a *Neighbourhood*.

6.4.1.2 Stencil and Properties

Stencil computations have been thoroughly investigated over the years with the development of image processing and numerical analysis. However, most libraries providing stencil skeletons assume, by default, that data collections are vectors or matrices. This is because, in practical problem domains, the data to be processed are typically stored as

vectors or matrices. Even parallel algorithmic skeleton library SkePU, which provides a limited number of smart containers, consider the data collection as by default a matrix or a vector. However, as introduced in the previous subsection, the definition of stencil computation does not necessarily require a grid-like data structure. Let us consider the example below,

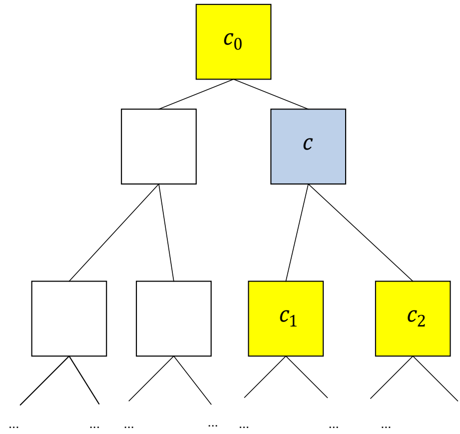


Figure 6.10: A stencil over a graph (tree)

In Figure 6.10, the data are stored in a graph structure, specifically in a tree data structure, where each node stores a single element of the data. Let us consider the stencil rule, where each element updates according to a function that takes all adjacent neighbours as parameters, e.g., blue node c are updated with all its neighbours, which are coloured by yellow.

Thus, we can conclude that *having a neighbourhood* is a *Necessary* property of a stencil skeleton, according to the definition provided for stencil skeletons and the illustration of an unconventional stencil in Figure 6.10. Now that we know we need to update an element with their neighbourhood elements, the question arises —how can we access other elements as we know they have neighbours? Therefore, *Iterable* should be another necessary property as we need to iterate the neighbour. Furthermore, *Ordered* should also be recognised as a necessary property of a stencil skeleton, since a collection in which each element is associated with its neighbours implies an internal order. Furthermore, to enable a parallel stencil, similar to `map` skeleton, *Random* is the *Sufficient* property. Thus, we can conclude that for a stencil skeleton,

Necessary properties- Iterable, *Neighbourhood*

Sufficient properties- Random

With the necessary and sufficient properties for a stencil skeleton defined, we can

further refine the model of a stencil skeleton to ensure consistent integration with the extended Collection Skeletons. Generally, a stencil kernel can be modelled as,

- A collection.
- A function to access the neighbours for each element.
- Border Policy.

Where BP (Border policy) can be regarded as the special case handling policy, and not every stencil kernel has a border policy. This model does not apply to every stencil kernel, however, it interprets a representative numbers of stencil kernels. To be compatible with the extended Collection Skeletons, we make the stencil signature as close as those data-centric algorithmic skeletons' as

```
Stencil(Collection, function, BP)
```

To further simplify the programming task for the programmer, we propose that when a stencil skeleton is invoked on a collection, the programmer should only need to focus on designing the stencil kernel and leave the remaining tasks to the Collection Skeletons, just as they do with the above data-centric algorithmic skeletons. Under the model introduced earlier, when programmers need to execute a stencil kernel on the data, they should first declare the data collection with the desired properties. Next, they must define the stencil rules within a function and design a border policy to address corner cases. After these steps, the programmer can invoke the stencil skeleton with all relevant parameters. Consequently, a stencil skeleton is thus produced and executed. In addition, the programmer does not need to concern themselves with the parallelisation of the stencil code; specifically, there is no need to manage for loops and border computations, as the Collection Skeletons implicitly handle these aspects. That is to say, the programmer only needs to explicitly invoke the stencil skeleton and design the rules for it; the Collection Skeletons can then implicitly perform the computations with high efficiency.

6.4.1.3 Implementation of Stencil

Based on our prototype library, we continue to implement both sequential and multi-threaded parallel versions of stencil skeletons to ensure compatibility with the prototype. The API of the prototype stencil is almost the same as the signature we just proposed,

```
Stencil(collection, function, (borderpolicy))
```

It should be noted that this signature encompasses only a limited subset of stencil kernels addressed in this work. The parallel version of the stencil skeleton is developed using Taskflow [70], which applies a task-based parallelisation framework to bring a multi-threaded performance boost to the stencil computation.

Moreover, we also provide compile-time property check for the stencil skeleton, ensuring compatibility with the declared properties of the data collection. As discussed earlier, the *Necessary* properties for a stencil are *Neighbourhood* and *Iterable*, while *Random* is the *Sufficient* property for a parallel stencil. The compile time property checking is also implemented following Algorithm 2.

Similarly, if a collection does not possess the *Necessary* properties, the compilation process will be interrupted, and an error message will be provided to the programmer for program correction. Conversely, if the collection satisfies the *Necessary* properties but lacks the *Sufficient* ones, the compilation will proceed, but with a warning to the user that the generated code will not be parallelised. This serves as a reference for programmers to decide whether to proceed with the current code or revisit their program to optimise for parallelism and achieve potential high performance. If the collection meets both the *Necessary* and *Sufficient* properties, a parallelised version of the stencil skeleton will be generated.

6.4.2 Wavefront

6.4.2.1 Introduction to Wavefront Skeleton

As introduced in Section 2.4.4.2, a `wavefront` skeleton implements a class of multi-dimensional recurrence relations that just like a sweeping of a wave, where in each iteration the diagonal is calculated as the front of the computation. Figure 6.11 presents a simple example for a wavefront skeleton that operates on a 4×4 matrix. In each iteration, which is represented by the blocks of the same colour, all elements in a diagonal can be computed simultaneously without data dependence as they only depend on the computational results from the previous diagonals. The direction of the wavefront as well as the computation is shown as the blue arrow beside the matrix. This exposes substantial opportunities to develop a parallel program for wavefront computation where the parallelism can also follow the wavefront of the computation.

Rectangular A collection is *Rectangular* if it has uniform row (and column) length and of regular shape, besides such uniformity and regularity needs to be consistent across its dimensions; otherwise, it is not *Rectangular*.

4×4 matrix

wavefront (computation) direction

Figure 6.11: Wavefront on a 4×4 matrix

6.4.2.2 Wavefront and Properties

Comparing to the stencil which has been widely used in numerous applications, wavefront is less popular and there has only been a little previous work covering wavefront. However, as the development of parallel computing architectures continue, with a proper computational abstraction, the domains covered by wavefront can see a boost. Thus, we investigate wavefront for its properties to integrate with the Collection Skeletons, and hopefully it can be better utilised based on the integration of both kinds of skeletons.

Similar to a stencil skeleton, when performing a wavefront computation along the diagonal of the collection, *Neighbourhood* should also be one of the *Necessary* properties —the elements of a wavefront need to access its neighbour elements to decide and update for the next wavefront. Besides, the wavefront needs to update the elements from the last computation thus they need to iterate over the elements, suggesting that *Iterable* should also be the necessary properties for a wavefront skeleton. Furthermore, a collection that can be operated upon by a wavefront needs to be *Ordered*, which is suggested by the aforementioned property *Neighbourhood*. As the description of wavefront shown in Figure 6.11, *Random* should be the *Sufficient* property that facilitate a parallel computation for a wavefront skeleton.

Most importantly, the nature of a wavefront requires the data collection to be *rectangular* or *square* to facilitate a wavefront skeleton. Therefore, we can conclude that to apply a wavefront (whether in parallel) on a data collection, the data collection should be,

- A rectangular or square collection;

- Have a function to guide the wavefront to update based on the neighbour of the current element.

Similarly, to simplify the efforts from the programmer's side and better integrate with the Collection Skeletons, we define a wavefront as,

```
Wavefront(Collection, function)
```

where the Collection is defined by the property-based abstractions introduced earlier and should satisfy the *Necessary* properties just mentioned, and *Sufficient* properties, if a parallel wavefront is required. Whenever the programmer needs to perform a wavefront operation on the data collection they declare with the properties, they only need to design the very details of the functions for each wavefront iteration, i.e., the wavefront and the wavefront update function.

6.4.2.3 Implementation of the Wavefront

As a proof-of-concept, we implement a wavefront skeleton sequential version and multi-thread parallel version to be compatible with the prototype library.

Based on the definition introduced just earlier, we design the signature of a wavefront to be as follows based on C++,

```
Wavefront(Collection, function)
```

where the function can be a function object or a lambda function as in C++. Similar to our prototype stencil implementation, we develop multi-threaded wavefront with Intel's oneAPI and Taskflow.

Similarly, based on the properties discovered for wavefront skeleton, we implement compile time properties checking for the input data collection. As discussed early, the *Necessary* properties for a `wavefront` are *Neighbourhood*, *Iterable*, *Rectangular* and *Random*, we adopt Algorithm 2 to perform property checking for `wavefront` as well. If the collection fails to meet the *Necessary* properties, the compilation will be interrupted and an error information will be returned to the programmer to help correct the program.

Wavefront is also suitable to be ported to heterogeneous platforms and porting it to GPU is a subject of our future work.

6.4.3 Divide-and-conquer

6.4.3.1 Introduction to Divide-and-conquer Skeleton

In contrast to `map` or `stencil` which are essentially data-centric algorithmic skeletons, divide-and-conquer is a task-oriented algorithmic skeleton, which divides a problem domain into several subproblem domains, i.e., subtask recursively until the innermost subtask and be easily conquered; then it combines the conquered results to get the final result for the original problem. Figure 2.9 presents an example for a divide-and-conquer skeleton. Ultimately, at the end of the dividing, each subtask can be performed with a "task", and all the results can be combined to the upper layer until the final result emerges.

6.4.3.2 Divide-and-conquer and Properties

As suggest by its name, if we would like to perform a divide-and-conquer over a collection, the collection firstly needs to be dividable, or as we have already specified in Chapter 4, *Splitable*, as is also shown in Figure 2.9. Thus, we define that *Spilitable* is the *Necessary* property for a `divide-and-conquer` over a collection.

However, the *Sufficient* properties for a parallel divide-and-conquer are hard to identify comparing to other algorithmic skeletons mentioned above. Whether a divide process can be performed simultaneously should be decided on multiple factors including the data dependence and the problem domain.

For example, Figure 6.12 applies a `divide-and-conquer` to sort a splitable collection of 4 integers. It firstly splits the collection into two halves, which is the divide step; then it recursively sorts both halves of the collection; after that, it merges the sorted halves to produce the sorted collection as a result. During the divide-and-conquer process, the recursive sorting of subsidiary collections in merge sort is independent and can be done in parallel, e.g., with multiple threads. However, the merge step would need synchronisation or a specific strategy to ensure results are combined in the correct order, which suggests that it can not be parallelised in some circumstances, or, at least not as easy as parallelised for the recursively sorting on subsidiary collections.

Thus, we currently do not include sufficient properties for a parallel divide-and-conquer and will investigate as future research.

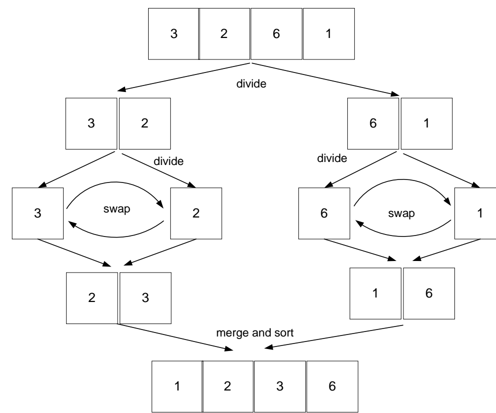


Figure 6.12: Merge sort using divide-and-conquer

6.4.3.3 Implementation of Divide-and-conquer

We implement a divide-and-conquer sequential version and extend the sequential version to a multi-thread parallel version.

Based on the definition introduced in the previous sections, we design the signature of a divide-and-conquer as,

```
Divide-and-Conquer(Collection, function)
```

where the function can be a function object or a lambda function in C++. As the complexity nature of a divide-and-conquer computation, our implementation only applies to a limited set of problems - those can be simply “divided” without processing many complex boundary cases. The sequential divide-and-conquer is implemented with the traditional recursive method. While for multi-threaded version we use Taskflow and Intel’s oneAPI to develop a task-based `divide-and-conquer`.

Based on the properties discovered for a divide-and-conquer skeleton, we implement a limited compile time properties checking for the input data collection as well. The prototype library involves checking the *Necessary* properties for a divide-and-conquer, which are *Splitable* and *Iterable*, based on Algorithm 2.

If the collection fails to meet the necessary properties of a `divide-and-conquer` skeleton, the compilation will be interrupted and an error information will be returned to the programmer to help correct the program.

Porting a `divide-and-conquer` skeleton to a GPU platform with the ability of properties checking against a collection with Collection Skeletons can be challenging as it has a nature of task-based processing, which is different from data-centric algorithmic skeletons that are suitable for processing on a GPU.

6.5 Evaluation

6.5.1 Evaluation of the Extension of Data-centric Algorithm Skeletons

We evaluate the prototype library with explicit parallelism against the 17 benchmarks programs as introduced in Chapter 4. Specifically, we evaluate data-centric algorithmic skeletons against the 17 benchmark programs as during the evaluation at Chapter 4, we found some structures can be substituted with data-centric algorithmic skeletons such as `map` and `reduce`. It is worthy to note that this process can also be performed by tools such as [16, 75]. The similar manually rewritten strategy has been applied —we manually investigate each of the 17 benchmark programs and explore if parallel patterns can be discovered, e.g., if a `map` algorithmic skeleton can be applied to substitute a segment, which is usually presented as a for loop in C++ program, in source code. Following the method in Section 4.6, we take the computational time of the rewritten programs with Collection Skeletons in Chapter 4 as the baseline, and calculate the speedup with the computational time of those rewritten with algorithmic skeletons in combination with Collection Skeletons.

During the manual investigation on the 17 benchmarks, we found 7 out of 17 benchmarks can be parallelised through explicit parallelism, i.e., by substituting a specific segment of source code with corresponding algorithmic skeletons while maintaining the functionality of each benchmark program. As before, a speedup greater than 1.0 indicates that the programs rewritten using both Collection Skeletons and Algorithmic Skeletons exhibit a performance improvement compared to the sequential versions; conversely, values less than 1.0 suggest a decline in computational performance. Table 6.2 presents the experimental results of the experiments, where algorithmic skeleton `for_each` can be viewed as another version of a `map` skeleton. Column *Details* show the substituted algorithmic skeletons for the corresponding benchmark program, e.g., for the benchmark program kmeans, `map` and `reduce` are applied to replace the for loops on centre calculation for the kmeans algorithm. The three columns next to column *Details* present the speedups for the same three platforms. As before, for circumstances where the parallelised version experiences substantial performance decrease, i.e., decreasing more than a factor of 0.01, we do not present the speedup in the table; instead, those results are denoted as < 0.01. Speedup results that are denoted as *N/A* mean those after-parallelised programs failed to compile or run, necessitating further investigation.

Speedups that are greater than 1 are highlighted for a better view. Similar to Section 5.4, SMT has been enabled on all three platforms without limitations to fully exploit multi-threaded computation. That is, we maintained the same thread configuration across all three platforms as specified in Table 5.2.

Table 6.2 presents the experimental results, where 2 out of 7 benchmark programs report speedup across all the three platforms, 4 out of 7 benchmark programs report speedup on one or two platforms, only 1 benchmark program reports performance decrease on all the three platforms. Most cases show moderate speedup with a factor less than 2, however, there are cases where the speedup is up to 32.22. Given the fact that we have only performed simple code substitution without extensively optimising the sequential code, the experimental results are promising. Compared to those algorithmic skeleton libraries introduced in Section 2.4, Collection Skeletons provide assurance for the programmers that they do not need to consider how to adapt their data collection for the algorithmic skeletons, as these details are hidden by the Collection Skeletons.

For the ising benchmark, the speedup by algorithmic skeleton is substantial as it reaches a factor of 32.22 on the Intel Server. The speedup is not surprising as 1. Ising benchmark is a "micro" benchmark with a simple control flow, thus it is more feasible by rewriting with algorithmic skeletons (`for_each`); 2. the `for_each` (`map`) skeleton tries to exploit all the cores of the CPU as it spawns to multiple threads. For the benchmark tinn, the computational performance is slightly worse than the baseline; It is worthy to note that explicit parallelisation performs the same as that substitution by only concurrent data structure as is shown in Table 5.1, while worse than that substitution by concurrent data structures with OpenMP directives as is shown in Table 5.2, for benchmark tinn. The result of the benchmark tinn is still reasonable, as algorithmic skeletons not necessarily performs better than implicit parallelism. The explicitly parallelised simpleHash fails to run on Intel Desktop and Intel Server, but reports a speedup of 1.78 on the Arm Server. This might have something to do with the architecture, but should be investigated further for an appropriate answer.

Furthermore, for Collection Skeletons, it is also possible to choose an optimal platform given the program source code, i.e., the property-based specification for the Collections with their corresponding algorithmic skeletons. For instance, in Table 6.2, the mri-q benchmark reaches better speedup on the Intel Sever; to achieve the better computational performance, it is possible to select the Intel Server as the target platform with the single piece of program. Similar to the flexibility introduced in Chapter 4, the selection of target platforms based on the property-based collections and the associated

algorithmic skeletons is subject to our future research.

Table 6.2: Parallelisation of the benchmarks (Explicit)

Benchmark	Details	Speedup Intel Desktop number of threads=12	Speedup Arm Server number of threads=4	Speedup Intel Server number of threads=72
ising	for_each	5.04	4.53	32.22
libactor	for_each	1.03	0.95	1.13
tinn	map, reduce, zip	0.95	1.01	1.00
simpleHash	for_each	N/A	1.78	N/A
lud	map reduce	0.14	0.07	< 0.01
kmeans	map reduce	1.10	0.99	0.97
mri-q	map reduce zip	1.12	1.09	1.17

6.5.2 Evaluation on Stencil Skeleton

In this section, we evaluate stencil skeleton, one of the advanced algorithmic skeletons introduced earlier in this work. We leave the evaluation of wavefront and divide-and-conquer for our future research.

We picked jacobi-1d and jacobi-2d [146] as the benchmark programs for this evaluation as jacobi computation involves a common stencil kernel. Most importantly, we need to evaluate the overhead and performance introduced by the Collection Skeletons, thus we need to calculate the speedup of the rewritten program against the original benchmark. To facilite such evaluation, we continue to follow the same methodology described in Chapter 4, we have rewritten the original jacobi-1d and jacobi-2d with our Collection Skeletons and associated algorithmic skeleton, the stencil. Listing 6.2 and Listing 6.3 below present the core source code of the original jacobi-1d program and the rewritten version with the skeleton strategy.

```
    static
void kernel_jacobi_1d(int tsteps,
                                   int n,
                                   DATA_TYPE POLYBENCH_1D(A,N,n),
                                   DATA_TYPE POLYBENCH_1D(B,N,n))
{
  int t, i;
#pragma scop
```

```
  for (t = 0; t < _PB_TSTEPS; t++)
    {
      for (i = 1; i < _PB_N - 1; i++)
        B[i] = 0.33333 * (A[i-1] + A[i] + A[i + 1]);
      for (i = 1; i < _PB_N - 1; i++)
        A[i] = 0.33333 * (B[i-1] + B[i] + B[i + 1]);
    }
#pragma endscop

...
//initialization of array A and array B
...
kernel_jacobi_1d(tsteps, n, POLYBENCH_ARRAY(A),
   POLYBENCH_ARRAY(B));
...
}
```

Listing 6.2: The original jacobi-1d program

```
//declaration and initialization of collections A, B and ce
   with properties
...

//define stencil rule through function jacobi_1d
stencil_dic jacobi_1d(){
    return stencil_dic{-1, 0, 1};
}

//define border policy bp
BorderPolicy bp;

//invoke the stencil skeleton
stencil(A, B, ce, jacobi_1d, bp, TSTEPS);
 ...
```

Listing 6.3: Jacobi-1d skeleton

```
#include<omp.h>
template<typename CollectionM, typename CE, typename Func>
```

```
void stencil(CollectionM &collection_A, CollectionM &
    collection_B, CE ce, Func f, BorderPolicy bp, int t){
     int x_max = collection_A.size();
     auto temp = f();
     int x = temp.x;
     int y = temp.y;
     int z = temp.z;
#pragma omp parallel for
     for(int k = 0; k < t; k++) {
          for (int i = 1; i < x_max; i++) {
               collection_B[i] = ce*(collection_A[i+x]+
                   collection_A[i+y]+collection_A[i+z]);
          }
          for (int i = 1; i < x_max; i++) {
               collection_A[i] = ce*(collection_B[i+x]+
                   collection_B[i+y]+collection_B[i+z]);
          }
     }

}
```

Listing 6.4: (OpenMP) implementation of a stencil skeleton

Compared to the original jacobi-1d code, the rewritten version introduces a stencil skeleton over collections `A` and `B`, and a coefficient collection `ce`. The programmer only needs to specify a *jacobi-1d* algorithm to describe the stencil computation and a border policy structure (`bp`). This not only simplifies the programming task, but also brings potential for portability and high performance computation across different platforms because of the integration of Collection Skeletons and algorithmic skeletons. Listing 6.4 shows the OpenMP implementation of a stencil skeleton that can execute in parallel. It is important to note that the * operator has been overloaded to perform multiplication between the coefficients collection and the neighboring elements. Moreover, the border policy `bp` is not utilised in Listing 6.4. Similarly, sequential versions of the stencil kernels can be obtained by disabling OpenMP multithreading.

Apart from that, there is no other code rewritten or optimisation as we aim for a like-for-like comparison. For the computational performance, we evaluate the sequential performance and parallel performance of our implementation separately. The computational environment configuration is the same as described in Chapter 4, where

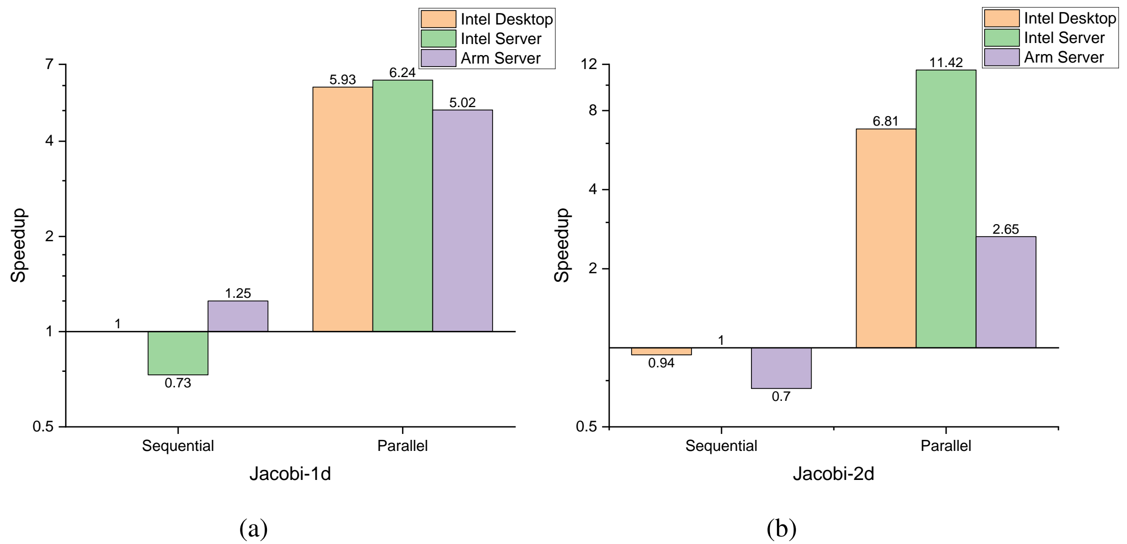
we accumulate the speedup for sequential stencil over the original one, and the parallel stencil over the original one, separately. For the parallel performance evaluation, as our prototype provides multi-thread implementation parallelised stencil, we still maintained the configuration for the number of threads used by OpenMP as specified in Table 5.2. Similar to the evaluation in Chapter 4, in Figure 6.13, a speedup greater than 1.0 indicates that the programs, either sequential or parallel, rewritten using Collection Skeletons and Algorithmic Skeletons exhibit a performance improvement compared to the original versions; conversely, values less than 1.0 represent a decline in computational performance.

Figure 6.13 presents the results for the evaluation, where the subsidiary figure (a) shows the speedup for evaluation on jacobi-1d benchmark and the subsidiary figure (b) shows the speedup for evaluation on jacobi-2d benchmark. For a more intuitive view on the speedup, bars below speedup line 1 show a decline of computational performance, while bars above speedup line 1 show an increase of computational performance.

For our sequential implementation of a stencil, there is slight variation for both benchmarks. This variation is acceptable as the underlying data structure has been changed from the original pointer-based array to an STL vector thus there might be minor performance variation when other structures stay the same for both programs; in consideration of our implementations and the rewritten methodology, and depending on the platforms as well as the data (e.g., the scale or the dimensions of the data), the results' variation is consistent with those shown in Figure 4.19.

For parallel cases for both benchmarks, substantial speedup has been reported from all the three tested platforms. For benchmark jacobi-1d, the speedup by parallel stencil is between 5.02 to 6.24 across three platforms; For benchmark jacobi-2d, the speedup by parallel stencil is between 2.85 to 11.42. There is a tendency that the more parallel cores the better the computational performance; however, we need more data to justify this in our future research. The experimental results suggest that by applying parallel stencil together with Collection Skeletons, the performance can be greatly improved with minimum efforts of code rewriting from the user's side.

The performance boost can also be scaled to other common stencils where matrix is the primary data structure, as the underneath data structure is similar to those used in our benchmarks (1-d vector and 2-d vector). However, for stencils against graph data collections, the performance scaling is unknown and subject to our future research.



(a) (b)

Figure 6.13: Speedup of Jacobi-1d and Jacobi-2d on three platforms

6.6 Chapter Summary

In this chapter, we have introduced explicit parallelism and the integration between algorithmic skeletons and our Collection Skeletons. We have proposed *Necessary* properties and *Sufficient* properties to help identify what properties a collection needs to be operated upon by an algorithmic skeleton, which is the foundation for a compile time property check for both categories of skeletons, making the bonding between both skeletons even more practical. We have also further introduced a couple of data-centric parallel algorithmic skeletons and selective advanced algorithmic skeletons based on Chapter 2. For a proof-of-concept purpose, we implemented prototypes for those algorithmic skeletons as well as integrating them with our Collection Skeletons based on our proposed prototype library. For the advanced algorithmic skeletons, we evaluated our stencil against two benchmark programs for its computational performance. The experimental results show that our sequential version of stencil skeleton shows little performance variation compared to the original ones, while the multi-threaded parallel version of stencil reports great speedup on the three tested platforms, suggesting the advantage of the integration of both categories of skeletons. Thus, programmers can write parallel code without much effort while keeping the computational performance with our Collection Skeletons and their integration with algorithmic skeletons.

It is essential to highlight that current implementation can only provide basic GPU support for the introduced algorithmic skeletons. For example, it is not possible to tune the implemented data-centric algorithmic skeletons for better performance

when running on GPUs. When attempting to executing two `maps` on a GPU, the computational performance will be slow as current implementation does not support skeletons fusion. Besides, the coverage of the stencil skeleton is limited as it only supports the most common stencil kernels such as jacobi or seidel. Although we have introduced parallelisation over distributed memory, we do not yet support either implicit parallelism or explicit parallelism across distributed nodes.

Chapter 7

Conclusion

7.1 Summary

Non-specialised programmers often encounter difficulties in efficiently writing programs that perform efficiently —specifically, when attempting to store data in a container, selecting the optimal one is non-trivial, leading to either underspecification or overspecification of the data collections. Overspecification not only results in inefficient programs but also impedes parallelisation, which is crucial for leveraging the multi-core or multi-threaded capabilities provided by contemporary hardware.

To assist programmers, particularly those with limited, in selecting the appropriate data collection for their specific problem domain, we propose a higher-level abstraction: Collection Skeletons. Rather than employing a hierarchical organization for data collections that focuses on implementation, we introduce a declarative approach that exposes essential properties of collections to programmers. This method facilitates a "flat" interaction paradigm, where programmers engage directly with the declaration rather than with the underlying complexity. By specifying data collections with their semantics and interface properties, the programmers only need to think about how the data collection should behave for their problem domain given the data to store, rather than knowing the fundamental implementation details before using it. We have prototyped the Collection Skeletons with C++ metaprogramming as well as the concrete data structure deduction algorithm. We show that our property-based approach to collections does not introduce performance overhead, but instead opens up the opportunity for performance improvements gained through greater implementation flexibility, which is hidden from the application programmer. In addition, based on the base version for Collection Skeletons and the prototype library, we show how parallelism can be exploited by two

strategies: Implicit Parallelism within parallel data access operations on concurrent data structures hidden from the programmer, and Explicit Parallelism exposed through parallel algorithmic skeletons under control of the application programmer. We have evaluated both parallelisation extensions for the prototype library separately on the same benchmark programs following the same methodology as Chapter 4, and concluded that both parallelism help the programmers write parallel programs while exposing opportunities for future performance optimisation.

7.2 Reflection

We propose implicit parallelism in Chapter 5, however the prototype library does not include all the features of the implicit parallelism as discussed. The computational performance benefit from implicit parallelism highly depends on proper implementations for parallel access functions, which we have yet to support. The performance provided by a concurrent data structure as implementation is limited. It could be better if we have introduced another concurrent data structure which comes with parallel access functions, and that we believe the performance could be greatly improved.

The interaction with the properties can be modified to enhance convenience. At present, we have classified properties into three categories. However, when a data collection is declared with these properties, the categories are not represented in the declaration as all the properties are put into the API together. It might have been advantageous to design a rule where the semantics and interface interact concerning the declaration of a data collection or are even involved in deciding on a concrete data structure.

7.3 Conclusion

It can be concluded that by introducing property-based abstractions for data collections, programmers especially non-specialised programmers can benefit from the abstract declarative interaction in aspects of programming efficiency and computational performance. Furthermore, the flexibility of the Collection Skeletons opens up more opportunities for further integration and expansion, e.g., with data structures from broader sources.

By introducing Implicit Parallelism and Explicit Parallelism, non-specialised programmers can develop parallel program easily with Collection Skeletons without need-

ing to worry about the overspecification or misspecification issue. Although the computational performance provided by both parallelism are still limited as of the evaluation on the prototype, it introduces great opportunities for future optimisation regarding numerous factors including the computational performance and programmability.

7.4 Future Work

In our future work, we will continue to include more properties into Collection Skeletons to improve the concept. Moreover, non-functional properties such as time complexity will be integrated into the Collection Skeletons to bring enhanced programmability to the programmers. We intend that the Collection Skeletons will cover as many concrete data structures as possible, which will eventually be a solid alternative to the traditional interaction with data structures.

Besides, we will continue to explore the interaction of our Collection Skeletons and algorithmic skeletons, opening up further performance benefits from parallelisation for more complex scenarios. We are exploring how to provide to support Collection Skeletons on heterogeneous platforms, e.g., GPUs to leverage the greater parallel performance of those platforms.

For the Collection Skeletons, the current prototype implementation and the multi-staged pattern matching algorithm are limited for more complex problem domains or larger size of data. Thus, we will also explore avenues to make concrete data structure selection more adaptive to the target machine and application context, e.g., through the use of machine learning methods for the selection of an optimal implementation data structure at runtime.

Furthermore, to enhance the expressiveness of the Collection Skeletons, we will explore implementing them as a DSL to create a data collection-oriented language. This approach will remove the limitations of C++ templates, making the Collection Skeletons even more convenient for programmers.

The Collection Skeletons are novel abstractions for data collections and are designed to be flexible for further extension, and more future work can be done based on this concept.

Bibliography

[1] Chirag Agrawal. Kruskal disjoint. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-using-stl-in-c/, 2021. [Accessed 20-Feb-2023].

[2] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. Accelerating Code on Multi-cores with FastFlow. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 170–181, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, Oct 2009.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] Benjamin Baka. *Python Data Structures and Algorithms: Improve application performance with graphs, stacks, and queues*. Packt Publishing Ltd, 2017.

[6] Blaise Barney. Introduction to parallel computing tutorial — HPC @ LLNL — hpc.llnl.gov. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial. [Accessed 06-10-2023].

[7] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T Barr. Darwinian data structure selection. In *Proceedings of the 2018 26th ACM Joint Meeting on*

European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 118–128, 2018.

[8] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 105–118, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Nathan Bell and Jared Hoberock. Chapter 26 - thrust: A productivity-oriented library for cuda. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359–371. Morgan Kaufmann, Boston, 2012.

[10] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eSkel. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, pages 761–770, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[11] Céline Bensoussan, Matthias Schöttle, and Jörg Kienzle. Associations in MDE: a concern-oriented, reusable solution. In *European Conference on Modelling Foundations and Applications*, pages 121–137. Springer, 2016.

[12] Manish Bhojasia. Joseph. `https:www.sanfoundry.com/c-program-solve-josephus-problem-using-linked-list`, 2013. [Accessed 20-Feb-2023].

[13] Aart J. C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. *Concurrency: Practice and Experience*, 9(6):579–619, 1997.

[14] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 68–78, New York, NY, USA, 2008. Association for Computing Machinery.

[15] Benjamin Brock, Scott McMillan, Aydın Buluç, Timothy G. Mattson, and José E. Moreira. C++ and interoperability between libraries: The GraphBLAS C++

specification. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 207–215, 2023.

[16] C. Brown, V. Janjic, A. Barwell, J. Thomson, R. Castañeda Lozano, M. Cole, B. Franke, J.D. Garcia-Sanchez, D. Del Rio Astorga, and K. MacKenzie. A hybrid approach to parallel pattern discovery in C++. In *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 187–191, 2020.

[17] Nick Brown. A programming model for developing application specific dataflow machines on FPGAs. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–1, 2022.

[18] Kim B Bruce. *Foundations of object-oriented languages: types and semantics*. MIT press, 2002.

[19] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. HELIX-RC: An architecture-compiler co-design for automatic parallelization of irregular programs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 217–228, 2014.

[20] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 84–93, New York, NY, USA, 2012. Association for Computing Machinery.

[21] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. The HELIX project: Overview and directions. In *DAC Design Automation Conference 2012*, pages 277–282, 2012.

[22] Simone Campanoni, Timothy M. Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. HELIX: Making the extraction of thread-level parallelism mainstream. *IEEE Micro*, 32(4):8–18, 2012.

[23] Martin Christopher Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines*. PhD thesis, Princeton University, USA, 1996. UMI Order No. GAX96-27387.

[24] Germán Castaño, Youssef Faqir-Rhazoui, Carlos García, and Manuel Prieto-Matías. Evaluation of Intel's DPC++ compatibility tool in heterogeneous computing. *Journal of Parallel and Distributed Computing*, 165:120–129, 2022.

[25] Venkatesan T. Chakaravarthy. New results on the computability and complexity of points–to analysis. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, page 115–125, New York, NY, USA, 2003. Association for Computing Machinery.

[26] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54. IEEE, 2009.

[27] Lin Chen, Di Wu, Wanwangying Ma, Yuming Zhou, Baowen Xu, and Hareton Leung. How C++ templates are used for generic programming: an empirical study on 50 open source systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(1):1–49, 2020.

[28] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[29] Steffen Christgau and Thomas Steinke. Porting a legacy CUDA stencil code to oneAPI. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 359–367, 2020.

[30] Brown Christopher, Barwell Adam D., Marquer Yoann, Céline Minh, and Zendra Olivier. Type-driven verification of non-functional properties. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, PPDP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Philipp Ciechanowicz, Michael Poldner, Herbert Kuchen, J Becker, K Backhaus, H L Grob, B Hellingrath, T Hoeren, Stefan Klein, Herbert Kuchen, U M Uller-Funk, Ulrich Wilhelm Thonemann, and G Vossen. Muesli - a comprehensive overview. Working Paper, 2009. European Research Center for Information Systems.

[32] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[33] Rosseta Code Contributors. Rosetta code — rosetta code. `https://rosettacode.org/wiki/Rosetta_Code`, 2022. [Accessed 20-Feb-2023].

[34] Marcin Copik and Hartmut Kaiser. Using SYCL as an implementation framework for HPX.Compute. In *Proceedings of the 5th International Workshop on OpenCL*, IWOCL 2017, New York, NY, USA, 2017. Association for Computing Machinery.

[35] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[36] Diego Costa and Artur Andrzejak. Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26, 2018.

[37] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 389–400, 2017.

[38] Thierry Cruanes, Benoit Dageville, and Bhaskar Ghosh. Parallel SQL execution in Oracle 10g. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’04, page 850–854, New York, NY, USA, 2004. Association for Computing Machinery.

[39] Xiaohui Cui, Frank Mueller, Thomas E. Potok, and Yongpeng Zhang. A programming model for massive data parallelism with data dependencies. In *Workshop on Programming Models for Emerging Architectures*, Parallel Architectures and Compilation Techniques (PACT), United States, Sep 2009. PARTICLE ACCELERATORS.

[40] Paweł Czarnul, Jerzy Proficz, and Krzysztof Drypczewski. Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. *Scientific Programming*, 2020:4176794, Jan 2020.

[41] Hércules Cardoso da Silva, Flávia Pisani, and Edson Borin. A comparative study of SYCL, OpenCL, and OpenMP. In *2016 International Symposium on Computer*

Architecture and High Performance Computing Workshops (SBAC-PADW), pages 61–66, 2016.

[42] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[43] Kaushik Datta, Mark Murphy, Vasily Volkov, Saneurl Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[44] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B Sartor, and Wolfgang De Meuter. Just-in-time data structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 61–75, 2015.

[45] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan 2008.

[46] Diego. GitHub - dlb04/infix-to-postfix ubinfix to postfix. — github.com. `https://github.com/dlb04/infix-to-postfix`, 2021. [Accessed 20-Feb-2023].

[47] Dobiasd. FunctionalPlus. `https://github.com/Dobiasd/FunctionalPlus`, 2024. Accessed: 2024-06-19.

[48] August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *International Journal of Parallel Programming*, 49(6):846–866, Dec 2021.

[49] Mokhtar Essaid, Lhassane Idoumghar, Julien Lepagnot, and Mathieu Brévilliers. GPU parallelization strategies for metaheuristics: a survey. *International Journal of Parallel, Emergent and Distributed Systems*, 34:497 – 522, 2019.

[50] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011.

[51] Jim Fisher. GitHub - airplug/Libactor: Actor Model Library for C — github.com. `https://github.com/airplug/libactor`, 2011. [Accessed 20-Feb-2023].

[52] Björn Franke, Zhibo Li, Magnus Morton, and Michel Steuwer. Collection Skeletons: Declarative abstractions for data collections. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, page 189–201, New York, NY, USA, 2022. Association for Computing Machinery.

[53] Björn Franke, Zhibo Li, Magnus Morton, and Michel Steuwer. Collection Skeletons: Declarative abstractions for data collections. Under Revision at *The Journal of Systems and Software*, `https://ssrn.com/abstract=4410575`, 2024.

[54] Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu. A survey of parallel sequential pattern mining. *ACM Trans. Knowl. Discov. Data*, 13(3), jun 2019.

[55] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 1–15, New York, NY, USA, 1996. Association for Computing Machinery.

[56] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, page 121–133, New York, NY, USA, 1998. Association for Computing Machinery.

[57] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting parallelism in c programs with recursive data structures. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, page 159–173, Berlin, Heidelberg, 1998. Springer-Verlag.

[58] Jay Godse. *Ruby Data Processing: Using Map, Reduce, and Select*. Apress, 2018.

[59] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[60] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 100–112, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Istvan Haller, Asia Slowinska, and Herbert Bos. Mempick: High-level data structure detection in C/C++ binaries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 32–41, 2013.

[62] Laurie J Hendren. Parallelizing programs with recursive data structures. Technical report, Cornell University, 1990.

[63] Laurie J Hendren and Guang R Gao. Designing programming languages for the analyzability of pointer data structures. *Computer Languages*, 19(2):119–134, 1993. ICCL '92.

[64] Laurie J. Hendren, Joseph Hummell, and Alexandru Nicolau. Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 249–260, New York, NY, USA, 1992. Association for Computing Machinery.

[65] L.J. Hendren and G.R. Gao. Designing programming languages for analyzability: a fresh look at pointer data structures. In *Proceedings of the 1992 International Conference on Computer Languages*, pages 242–251, 1992.

[66] L.J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[67] Christoph Armin Herrmann. *The skeleton based parallelization of divide and conquer recursions*. PhD thesis, Universitat, Passau, 2001.

[68] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. MapCG: Writing parallel program portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 217–226, New York, NY, USA, 2010. Association for Computing Machinery.

[69] Kuan-Chieh Hsu and Hung-Wei Tseng. Accelerating applications using edge tensor processing units. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[70] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Trans. Parallel Distrib. Syst.*, 33(6):1303–1320, Jun 2022.

[71] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, Sep 1992.

[72] Joseph Hummel, Laurie J. Hendren, and Alexandru Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 218–229, New York, NY, USA, 1994. Association for Computing Machinery.

[73] Joseph Hummel, Alexandru Nicolau, and Laurie J. Hendren. Applying an abstract data structure description approach to parallelizing scientific pointer programs. Technical report, University of California, 1992.

[74] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70, 2011.

[75] Vladimir Janjic, Christopher Brown, and Adam D. Barwell. Restoration of legacy parallelism in C and C++ applications. In *13th International Symposium on High-Level Parallel Programming and Applications (HLPP 2020)*, pages 134–153, 2020.

[76] Damian Jarek. GitHub - djarek/md5lamacz: A multithreaded brute-force MD5 hashed password cracker. — github.com. `https://github.com/djarek/md5lamacz`, 2015. [Accessed 20-Feb-2023].

[77] Innes Jelly and Ian Gorton. Software engineering for parallel systems. *Information and Software Technology*, 36(7):381–396, 1994. Software Engineering for Parallel Systems.

[78] Zheming Jin and Hal Finkel. A case study of k-means clustering using SYCL. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4466–4471, 2019.

[79] Changhee Jung and Nathan Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 56–66, New York, NY, USA, 2009. Association for Computing Machinery.

[80] Changhee Jung, Silvius Rus, Brian P Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. *ACM SIGPLAN Notices*, 46(6):86–97, 2011.

[81] Sam Kamin. Some definitions for algebraic data type specifications. *SIGPLAN Not.*, 14(3):28–37, mar 1979.

[82] Chuanle Ke, Lei Liu, Chao Zhang, Tongxin Bai, Bryan Jacobs, and Chen Ding. Safe parallel programming using dynamic dependence hints. *SIGPLAN Not.*, 46(10):243–258, Oct 2011.

[83] Alain Ketterlin and Philippe Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 437–448, 2012.

[84] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Task parallelism and data distribution: An overview of explicit parallel programming languages. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, pages 174–189, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[85] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Prospector: A dynamic Data-Dependence profiler to help parallel programming. In *HotPar'10: Proceedings of the USENIX workshop on Hot Topics in parallelism*, Berkeley, CA, June 2010. USENIX Association.

[86] Wooyoung Kim and Michael Voss. Multicore desktop programming with Intel threading building blocks. *IEEE Software*, 28(1):23–31, 2011.

[87] Sandeep Koranne. *Boost C++ Libraries*, pages 127–143. Springer US, Boston, MA, 2011.

[88] Chris Lattner and Lauro Venancio. test-suite/SingleSource/Benchmarks/Shootout/hash.c at master · llvm-mirror/test-suite — github.com. `https://github.com/llvm-mirror/test-suite/blob/master/SingleSource/Benchmarks/Shootout/hash.c`, 2021. [Accessed 20-Feb-2023].

[89] Zhen Li, Ali Jannesari, and Felix Wolf. An efficient data-dependence profiler for sequential and parallel programs. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 484–493, 2015.

[90] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, page 50–59, New York, NY, USA, 1974. Association for Computing Machinery.

[91] Barbara Liskov and Stephen Zilles. Specification techniques for data abstractions. In *Proceedings of the International Conference on Reliable Software*, page 72–87, New York, NY, USA, 1975. Association for Computing Machinery.

[92] Steven F Lott. *Functional Python programming: Discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads*. Packt Publishing Ltd, 2018.

[93] Gustav Louw. GitHub - glouw/Tinn: A tiny neural network library — github.com. `https://github.com/glouw/tinn`, 2020. [Accessed 20-Feb-2023].

[94] D. Maley, P. L. Kilpatrick, E. W. Schreiner, N. S. Scott, and G. H. F. Diercksen. The formal specification of abstract data types and their implementation in Fortran 90: implementation issues concerning the use of pointers. *Computer Physics Communications*, 98(1):167–180, Oct 1996.

[95] Jason Marcell. GitHub - jasmarc/scheduler: CPU scheduling simulator — github.com. `https://github.com/jasmarc/scheduler`, 2009. [Accessed 20-Feb-2023].

[96] Stefan Marr and Benoit Daloze. Few versatile vs. many specialized collections: How to design a collection library for exploratory programming? In *Companion Proceedings of the 2nd International Conference on the Art, Science, and Engineering of Programming*, Programming '18, page 135–143, New York, NY, USA, 2018. Association for Computing Machinery.

[97] Martin Odersky and Lex Spoon. The architecture of scala collections. `https://docs.scala-lang.org/overviews/core/architecture-of-scala-collections.html`, 2019. [Accessed: 01-Oct-2023].

[98] Millán A. Martínez, Basilio B. Fraguela, and José C. Cabaleiro. A highly optimized skeleton for unbalanced and deep divide-and-conquer algorithms on multi-core clusters. *The Journal of Supercomputing*, 78(8):10434–10454, May 2022.

[99] Tim Mattson and Michael Wrinn. Parallel programming: Can we please get it right this time? In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, page 7–11, New York, NY, USA, 2008. Association for Computing Machinery.

[100] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. Dawncc: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2), May 2017.

[101] Scott Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*. Pearson Education, 2001.

[102] Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Comput. Surv.*, 47(4), Jul 2015.

[103] Siddharth Mohanty and Murray Cole. Autotuning wavefront abstractions for heterogeneous architectures. In *2012 Third Workshop on Applications for Multi-Core Architecture*, pages 42–47, 2012.

[104] Siddharth Mohanty and Murray Cole. Autotuning wavefront applications for multicore Multi-GPU hybrid architectures. In *Proceedings of Programming*

Models and Applications on Multicores and Manycores, PMAM'14, page 1–9, New York, NY, USA, 2018. Association for Computing Machinery.

[105] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, 2009.

[106] Maurice Naftalin and Philip Wadler. *Java Generics and Collections*. O'Reilly, May 2006.

[107] Quan M. Nguyen and Daniel Sanchez. Pipette: Improving core utilization on irregular applications through intra-core pipeline parallelism. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 596–608, 2020.

[108] Quan M. Nguyen and Daniel Sanchez. Phloem: Automatic acceleration of irregular applications with fine-grain pipeline parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1262–1274, 2023.

[109] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, Mar 2008.

[110] Daniel E. O'Leary. Artificial Intelligence and Big Data. *IEEE Intelligent Systems*, 28(2):96–99, 2013.

[111] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[112] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[113] Norbert Pataki. Testing by C++ template metaprograms. *arXiv e-prints*, 2010.

[114] Varsha H Patil. *Data Structures using C++*. Oxford University Press, Inc., 2012.

[115] Pierre Paulin. Programming challenges & solutions for multi-processor SoCs: An industrial perspective. In *Proceedings of the 48th Design Automation Conference*, DAC '11, page 262–267, New York, NY, USA, 2011. Association for Computing Machinery.

[116] Michael Perrone. Multicore programming challenges. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 1–2, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[117] Ivanilton Polato, Reginaldo Ré, Alfredo Goldman, and Fabio Kon. A comprehensive view of Hadoop research—a systematic literature review. *Journal of Network and Computer Applications*, 46:1–25, 2014.

[118] Rafa32. GitHub - rafa32/Quantum-Shor: Implementation of a quantum computer simulator together with Shor's algorithm. — github.com. `https://github.com/rafa32/Quantum-Shor`, 2017. [Accessed 20-Feb-2023].

[119] James Reinders, Michael Voss, Pablo Reble, and Rafael Asenjo-Plaza. C++ for heterogeneous programming: oneAPI (DPC++ and oneTBB). `https://www.oneapi.io/c-for-heterogeneous-programming-oneapi-dpc-and-onetbb/`, Nov 2020. Accessed: 2023-10-28.

[120] Gabriel Rodriguez-Canal, Nick Brown, Yuri Torres, and Arturo Gonzalez-Escribano. Programming abstractions for preemptive scheduling on FPGAs using partial reconfiguration. In Jeremy Singer, Yehia Elkhatib, Dora Blanco Heras, Patrick Diehl, Nick Brown, and Aleksandar Ilic, editors, *Euro-Par 2022: Parallel Processing Workshops*, pages 133–144, Cham, 2023. Springer Nature Switzerland.

[121] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Hideki Saito, Rakesh Krishnaiyer, Mikhail Smelyanskiy, Milind Girkar, and Pradeep Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? *SIGARCH Comput. Archit. News*, 40(3):440–451, Jun 2012.

[122] Miriam Schmidberger and Bernd Brügge. Need of software engineering methods for high performance computing applications. In *2012 11th International Symposium on Parallel and Distributed Computing*, pages 40–46, 2012.

[123] Matthias Schöttle and Jörg Kienzle. On the difficulties of raising the level of abstraction and facilitating reuse in software modelling: the case for signature extension. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*, pages 71–77. IEEE, 2019.

[124] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for gadts. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, page 341–352, New York, NY, USA, 2009. Association for Computing Machinery.

[125] Saumyendra Sengupta and Carl P Korobkin. *C++: object-oriented data structures*. Springer, 2012.

[126] Gleison Souza Diniz Mendonça, Breno Campos Ferreira Guimarães, Péricles Rafael Oliveira Alves, Fernando Magno Quintão Pereira, Márcio Machado Pereira, and Guido Araújo. Automatic insertion of copy annotation in data-parallel programs. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 34–41, 2016.

[127] Alexander Stepanov and Lee Meng. The Standard Template Library. Technical Report HPL-95-11(R.1), HP Laboratories, 1995.

[128] Michel Steuwer, Michael Haidl, Stefan Breuer, and Sergei Gorlatch. High-level programming of stencil computations on Multi-GPU systems using the SkelCL library. *Parallel Processing Letters*, 24(03):1441005, 2014.

[129] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127:27, 2012.

[130] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL parallel container framework. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 235–246, New York, NY, USA, 2011. Association for Computing Machinery.

[131] Peter Thoman, Florian Tischler, Philip Salzmann, and Thomas Fahringer. The celerity high-level API: C++20 for accelerator clusters. *International Journal of Parallel Programming*, 50(3):341–359, Aug 2022.

[132] P. W. TRINDER, K. HAMMOND, H.-W. LOIDL, and S. L. PEYTON JONES. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[133] Chih-Fong Tsai, Wei-Chao Lin, and Shih-Wen Ke. Big Data mining with parallel computing: A comparison of distributed and mapreduce methodologies. *Journal of Systems and Software*, 122:83–92, 2016.

[134] Yu-Hsiang M. Tsai, Terry Cojean, and Hartwig Anzt. Providing performance portable numerics for Intel GPUs. *Concurrency and Computation: Practice and Experience*, 35(20):e7400, 2023.

[135] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 49–59, 2007.

[136] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

[137] Chris Vasiladiotis. mischung-suite/programs/ising/data/original_-source/ising.c at master · compor/mischung-suite — github.com. https://github.com/compor/mischung-suite/blob/master/programs/ising/data/original_source/ising.c, 2020. [Accessed 20-Feb-2023].

[138] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, page 79–89, New York, NY, USA, 2007. Association for Computing Machinery.

[139] David W Walker and Jack J Dongarra. MPI: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[140] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. Complexity-guided container replacement synthesis. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1):1–31, 2022.

[141] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. Tree dependence analysis. *SIGPLAN Not.*, 50(6):314–325, Jun 2015.

[142] Timmy Whelan. Mp3. `https://sourceforge.net/projects/mp3reorg/files/mp3reorg/`, 2002. [Accessed 20-Feb-2023].

[143] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023.

[144] Guoqing Xu. CoCo: Sound and adaptive replacement of java collections. In *European Conference on Object-Oriented Programming*, pages 1–26. Springer, 2013.

[145] Yang You, Haohuan Fu, Shuaiwen Leon Song, Amanda Randles, Darren Kerbyson, Andres Marquez, Guangwen Yang, and Adolfy Hoisie. Scaling support vector machines on modern hpc platforms. *Journal of Parallel and Distributed Computing*, 76:16–31, 2015. Special Issue on Architecture and Algorithms for Irregular Applications.

[146] Tomofumi Yuki. Understanding polybench/c 3.2 kernels. In *International workshop on polyhedral compilation techniques (IMPACT)*, pages 1–5, 2014.

[147] S. Zellmann and U. Lang. C++ compile time polymorphism for ray tracing. In *Proceedings of the Conference on Vision, Modeling and Visualization*, VMV '17, page 129–136, Goslar, DEU, 2017. Eurographics Association.