Parallel Algorithms and Architectures for VLSI Pattern Generation

Roderick David Wolfe Widdowson

Ph D

University of Edinburgh

1987

Abstract

The process of data preparation for pattern generation consists of the performing of various geometric operations upon a user's design. This is the final manipulation of the design prior to the making of the masks and can be exceptionally time consuming, especially when a very large user designs are being processed.

This thesis addresses one way of speeding up the processing - by the use of (relatively) small numbers of loosely coupled processors as a multiprocessing unit. The starting point of the work was commercial pattern generation system. This incorporated programs programs or modules to perform parsing and flattening, overlap removal (merging) and partitioning. The major results are a series of performance measurements obtained by emulating parallel processors with an appropriate interconnection pattern. These give a valuable indication of the degree of performance enhancement which this approach to parallelism may be expected to bring. Confirmation of these expectations via practical implementations on two varieties of workstation clusters is reported.

The three major parts of a Pattern Generation system, the merge stage whose most important aspect is overlap removal and sizing, and the decomposition and sort (termed *fracture*) for the two major types of lithographical equipment (electron-beam and photo-mechanical) are examined separately. Particular emphasis is placed on the merge stage, which is the most time critical part of the whole process. The method used to partition the merge task among processors is area subdivision.

The practical work carried out consisted of devising practical algorithms for the area subdivision and recombination, and in the setting up of test harnesses multiprocessor emulations.

Acknowledgements

I should like to acknowledge the assistance of all the members of the Computer Science department but my particular thanks must go to David Rees who supervised this thesis.

Lattice Logic helped financially and the members of staff, both past and present, have been most supportive. In particular I thank John Gray for his constant encouragement. All the members of the ShapeSmith group, and in particular Neil Menzies and Gordon Hughes have helped me immensely to formulate and develop the ideas presented here.

This work was funded under the Cooperative Awards in Science and Engineering scheme by the United Kingdom Science and Engineering Research Council and Lattice Logic Ltd.

Finally I must thank my parents for their support over the years.

Declaration

I hereby declare that this thesis has been composed by myself, and that the work it describes is my own.

Roderick David Wolfe Widdowson

September 7, 1987

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the packing density, size and consequently the shape count of integrated circuits becomes greater, the time taken to do all the data processing associated with generating the designs is increasing. One particular process which is rapidly becoming a bottleneck is the data preparation prior to pattern generation.

Simultaneously the demand for high speed turnaround of IC designs is increasing. Technological improvements have reduced turnaround to the extent that IC design could now be described as an interactive rather than batch process, by analogy to the technological improvements to operating systems which allowed the same transition to happen to programming.

This thesis addresses the problems of improving the performance of pattern generation data preparation through the exploitation of parallelism and discusses the software engineering issues involved in parallelising pattern generation algorithms.

1.1 Making Programs Run Faster

As soon as the earliest computers had been built and demonstrated to work, the race to develop faster machines started. Improvements have been made by speeding parts from the complete range of the system hierarchy of the system, from the lowest level of hardware to the highest level of theoretical algorithmic design. For the purposes of further explanation it is useful to divide this hierarchy into three broad categories. The choice of three categories is arbitrary since the hierarchy of a computer system is continuous and so the actual boundaries between categories are somewhat fluid. The categories are: the basic algorithm, its software implementation and its hardware implementation; each of these is amenable to improvement,

1.1.1 Algorithmic improvement

The algorithmic theorist, when looking for ways to improve systems, develops better algorithms in terms of *asymptotic* performance - that is the rate at which the algorithm gets slower as the input gets larger. Algorithms are independent of the technology of their implementation and of the computers upon which they run.

Algorithmic improvements are more important than technological ones. Consider for instance a system which requires an unordered set of data to be sorted according to some key. There are many sorting algorithms, but consider just two - insertion sort and merge sort. The time complexities of these are $O(n^2)$ and $O(n \log n)$ respectively. The former is a somewhat simpler algorithm and for small input data sizes might be faster. However there is a value for n for which the latter will be faster. This is always the case no matter what the implementation is - there will always be a value for n at which a mergesort running on the slowest hardware with an inefficient implementation in software is faster than an insertion sort on the fastest possible hardware with a 'perfect'

software implementation. Algorithmic development is therefore a vital first step in the speeding up of processing.

In some problems the output of a program may be correct (in some way) but are not the theoretical best or minimum. For instance a program may generate solutions to the travelling salesman problem which are feasible (and effectively usable), but may not guarantee to generate the *minimal* solution. Very often problems whose minimal solution is *NP* complete can be approximated to by use of polynomial algorithms.

1.1.2 Improving the Software Implementation

The software engineer looks for ways to improve system performance by investigating techniques which improve the implementation of the algorithm in software. This ensures that the algorithm and the rest of the software supports for the system are expressed in the best possible form for the subsequent interpretation by the hardware. Software engineering covers that range of the system hierarchy which relates to the expression and interpretation of an algorithm.

At the highest level of implementation the job of the software engineer is somewhat similar to that of the algorithmic theorist; however, the aims may be somewhat different and this is discussed further below. For instance the engineer might seek to improve the implementation by running 'profiling' on the implementation and, having noticed those parts which were exercised most often, making sure that their implementation is as good as possible. These changes could be as simple as the reordering of the evaluation of conditions, or might entail unwinding commonly used procedures 'in line'.

Situated in the hierarchy somewhat below the the engineer interested in individual implementations are those investigating improvements to the means of implementation (and thus many implementations). For instance the compiler engineer might be interested in improving the quality of object code which the compiler generates.

At the lowest level the software engineer might develop the microcode which is the interface between hardware and software.

1.1.3 Improving the Hardware Implementation

The hardware engineer may attempt to speed up a computing system by developing techniques which improve the basic hardware upon which the system is based. Again there is a broad range of options which are open to the hardware engineer. At the highest level there is no real boundary between the work of the software and the hardware engineer. At the lowest level the hardware engineer must be involved with the physics which govern electronics.

The majority of techniques used to speed up hardware use parallelism, these are detailed later, but the following points illustrate some of the options open to the hardware engineer over the hierarchy of hardware.

- increasing the basic clock speed of the hardware. If this is possible it immediately makes the complete system faster.

- Related to the above technique is speeding up the other parts of the system, for instance the mass storage system.

- The majority of programs show some sort of locality of reference. Introduction of cache memories exploits this locality both in the data and the instructions.

1.1.4 Assessing the performance improvement

All the techniques for performance improvements are independent of each other. Thus for instance any improvement achieved by a software engineer is totally orthogonal both to improvements made to any basic algorithm (in as much as this would not affect the software implementation), or other improvements made to the implementation by hardware or software engineers.

In all three cases, algorithmic, software, and hardware improvement, what is being attempted is a reduction of a ***cost function.*** The precise cost function differs between algorithmic theorists and those performing the actual implementation. Indeed it may be different between engineers working at separate parts of the design hierarchy. Specifically, in order to be able to argue rigorously, algorithmic theorists deal only in orders of growth, and fixed bounds to these orders of growth. This make is possible to classify the algorithm's behaviour as the input becomes larger and thus look for ways to reduce the *asymptotic* performance.

In implementation the cost function is both considerably different and rather more general. In speeding up implementation, techniques are normally aimed at making a system run faster *for a given size of input*, thus improving the *absolute* performance. Other points which need to be taken into account when examining techniques to improve implementation and hence form part of the total cost function are pragmatic issues such as cost, reliability and supportability.

Thus a theorist may develop an algorithm which is better, since it exhibits a lower order of growth. The software engineer may choose not to implement this algorithm since according to the cost function which the software engineer uses the algorithm is worse. This might be due to the intricacy of the algorithm which makes the absolute performance for small input unacceptable, or perhaps the algorithm is so complex that any implementation would be unmaintainable. Finally since a theoretically minimal solution to a problem may be infeasible computationally a software engineer will trade off 'goodness' of solution against computation time.

1.2 Parallelism

There are many techniques which may be applied at many places across the system design hierarchy. For instance many of the techniques developed to control the complexity of software design are equally applicable to controlling the complexity of Very Large Scale Integrated circuits.

One of the most useful techniques which can be used to increase the performance of computer systems is the harnessing of parallelism. It can be applied across the complete spectrum of the design hierarchy. Actual implementations have met with varying degrees of success. Parallelism may be studied by algorithmic theorists, software engineers and hardware engineers.

1.2.1 Algorithmic parallelism

The computational complexity of parallel systems has been studied in great depth and many very interesting algorithms and architectures have resulted.

When implementing these algorithms there is a fundamental problem, which springs from the fact that the cost functions used to analyse concurrent algorithms differ considerably from those used by engineers. Typically theoretic measurements of complexity are based on a measure not only of asymptotic time performance, but also hardware complexity. Time is open ended - thus it is reasonable to talk of asymptotic time performance. Hardware, however, is not and so any algorithm which assumes an open ended amount of hardware must have a limit on its size of input.

Thus although Batcher's bitonic sorting algorithm [BDHM84] has time complexity $O(\log^2 n)$, the fact that its processor complexity (the rate of growth of the number of processors) is $O(n)$ makes implementation difficult. Assume that a sorting engine was built to use bitonic sorting with k processors. Since the hardware size is fixed, once the input exceeds the limit of k processors the sorting engine ceases to function according to the algorithms and becomes purely a

faster machine and as such can only affect the absolute performance; for small inputs only a fraction of the available processing power is used.

This does not in any way decry the work done on algorithmic parallelism, it merely points out the problems of generally applying something which may appear to be good only within a limited cost function. The insights gained by theoretical analysis are of very great use. However, one of the limitations assumed in this thesis is that hardware is not unlimited. This thesis does not address algorithmic parallelism, it describes some algorithms which run on parallel architectures and achieve an absolute performance increase.

1.2.2 Implementing parallelism

Given the possibility of concurrent hardware, the software and hardware engineer would seek to find methods to improve the *absolute* performance. The resulting system would have no upper limit on the input size. All data, regardless of size, would be processed faster. The ideal engineering solution is to improve the absolute speed of a parallel system by a factor close to the increase in hardware complexity.

Concurrency has been implemented right across the design hierarchy. The earliest techniques developed for its implementation were aimed at the lowest levels of the design hierarchy. With a few remarkable exceptions, development has been of techniques which are a progression up the design hierarchy, through the hardware hierarchy to the lower levels of the software hierarchy where most research is currently taking place. The reason that the highest level of hierarchy (the theory of algorithms) has already a substantial amount of results is that the cost functions used allow mathematically rigorous argument about parallel systems.

In his stimulating article [Den86] Denning notes the progression up the design hierarchy and divides the development of the implementation of parallelism into four stages. It should be mentioned that, as with all form of artificially im-

posed division, there are some developments which defy classification. However for the most part this classification remains as being very useful.

Stage One is the explicit addition of parallelism to the hardware. This is done in such a way as not to require changes to be made to the users' code.

At the lowest level of the hardware hierarchy, what is being made concurrent is the action of individual gates and transistors. An obvious move is from bit serial to bit parallel processing. This change is more effective than the next 'obvious' move to word parallel processing since the unit of operation in most algorithms is the word, not groups of words.

That bit parallel processing is not used throughout all hardware design is an interesting example of the effect of cost-functions. In many cases, especially when extreme packing density of circuits is desired, the simplicity of bit serial processing makes them favoured over the complexities (and greater size) of bit parallel implementations.

An interesting development of such ideas is the bit serial SIMD machine such as the DAP [HJ81, section 3.3] . Instead of processing words serially and bits in parallel these machines function by processing bits serially and words in parallel. An earlier and somewhat more complex example is the orthogonal computer [Sho70] which allowed dual modes of operation - bit serial word parallel and *vice versa*. Both these machines are somewhat out of the progression being examined here since programming them requires explicit knowledge of the architecture and their specialised op-codes thus disqualifying them from pure classification under stage one or stage two. This indicates the weakness of the classification.

Another popular technique for enhancing the absolute processing throughput of a computing system is to incorporate various *pipelines* into it. Use of pipelines stems from the observation that very often access to memory is sequential in nature, allowing prediction of where the next access will be. This is especially true for the instruction streams which make up programs. Pipelined instruction execution units work by dividing the handling of instructions into

separate parts, each of which may be dealt with by a separate piece of hardware. Once the instruction has been processed by one piece of hardware, it moves along to the next site in the pipeline and the next instruction can then be processed. Thus several instructions are processed in parallel, although the fixed order of processing is maintained. Parallelism has allowed more processing power to be introduced without making changes to the users' original programs. Most modern CPUs have some level of instruction pipelining.

In a similar manner, pipelining can be applied to the data stream [HJ81, chapter 2], thus introducing parallelism. In practice, the number of applications which can easily and invisibly utilise data pipelined architectures is limited.

The best example of a data-pipelined architecture are the current 'supercomputers' for instance the Cray series machines. Programming these machines need not require explicit and detailed knowledge of the architecture. Vectorising compilers and special purpose run time libraries make use of the vector instructions which utilise the pipelines.

At the highest level of hierarchy which still remains within the bounds of stage one development are the loosely coupled multiprocessors such as the DEC Vax-Cluster[KLS86]. These usually operate in conjunction with a multitasking operating system. Each process is assigned to one processor, either for its 'life' or, rather more flexibly, for one time-slice. The available concurrency is used, not to speed up the execution of one task, but to allow processing of more than one task in parallel. Thus no single task is performed fast, but more processes can be performed in parallel without performance degradation. Again no change needs to be made to the users' code. Such hardware is often the only way, and is currently the most cost-effective way, of increasing processing power of multi-user installations. A useful side effect of this sort of hardware is the added reliability, caused by the inherent redundancy.

Stage Two. In this stage, the parallelism is explicitly introduced into the system at the software level. In terms of the design hierarchy, development

here is at the highest levels of hardware and the lower levels of software, comprising as it does of both software and hardware architecture. The software engineer explicitly divides the system into *processes* which are then assigned to processors. Any inter-processor communication needs to be explicitly handled.

It is in order to ease the handling of such concepts as multiprocessing and interprocessor communication that special purpose languages such as OCCAM have been developed. Although their use is not obligatory, as witnessed by this thesis, they will considerably ease the process of implementation. The drawback with using such languages at present is that they are exceptionally difficult to debug. With traditional languages it is normally very easy to detect when a program has failed. However, as well as the normal methods of failure, parallel languages can also fail due to deadlock whose detection is not trivial. Fairness (equal resource allocation) is another problem which these languages need to address. The further development of such languages will greatly ease further research during this stage.

Another highly significant recent development are the public domain Remote Procedure Call systems. Any (inherently parallel) program written using these to control the parallelism is guaranteed at least a certain amount of portability (being amongst those parallel systems upon which the RPC has been mounted). The wider availability of these systems and their general acceptance will lead the way to 'portable parallelism' just as the development of high level languages paved the way for portable programs.

Currently most target hardware for this stage is relatively coarse grain MIMD [Fly72] with a maximum of a few hundred processing units. Given the restrictions of Amdahl's law (see section 1.2.3) it is difficult to see how any larger system could be effectively programmed, except in a few very special purpose situations. The processing units are all interconnected. A common interconnection is the n-dimensional cubes (as used by the INTEL iPSC, a derivative of the Caltech Cosmic cube [Sei85] and NCUBE/ten [HMS*86]) although grids and shuffle exchange graphs are possible.

In as much as they may be placed in the classification, SIMD machines such as the aforementioned DAP may be placed in stage two by virtue of their explicit reference to parallelism. However they have not yet proved popular for implementation of stage two developments; this is probably due to the difficulty in envisaging algorithms for them. Most SIMD machines can be classified as belonging either to stage one although there are certain indications that they will play a role in stage three.

The work described in this thesis can be unequivocally placed into the second stage. The concurrency is explicitly described; this is done within the IMP language [Rob86], the parallelism being specified by a technique similar to remote procedure calls. The precise mechanism is described later.

Stage Three. From being explicitly described by the engineer, the parallelism becomes implicit in the language, the compiler and associated software extracts the concurrency from the language. The languages are likely to be non-imperative.

Whereas the second stage is currently the centre of investigation for the low to medium level software engineer, the third stage is currently the outpost of research at the highest levels of software engineering. No particular trend for target architectures for this stage of development has appeared yet; developments include dataflow machines [TBH82] and Logic In Memory machines such as the DAP and the Connection Machine.

Denning [Den86] infers that this stage will supercede stage two. This seems unlikely since it is more probable that each stage will develop techniques which will be, as all other techniques developed across the design hierarchy, orthogonal to each other. Thus any improvement made during stage two will be independent of changes made during stage three - just as the improved hardware developed during stage one can, and is, used as the base-level implementation for machine used in stage two.

Stage Four. Beyond saying that this will consist of

> "... very high-level user interfaces capable of interacting with scientists (and engineers) at the same level of abstraction as scientists do with each other ..."

Denning gives no details of how this rather sweeping aim may or will be achieved and it is safe to assume that research in this field is some way off.

Current Research into Parallelism

The great boost which research in concurrency has recently received is due to several factors. These have affected the cost functions which in turn have meant that concurrency has become a viable (and indeed a vital) method of developing faster hardware.

- The cost of hardware has reduced significantly while the reliability has increased markedly. This is in contradistinction to software which is currently the major cost factor in a system while also representing the greatest source of unreliability.

- Many other implementation levels are approaching their cost-effective limits eg basic device speed. In order to win further performance increases different methods need to be developed to improve system performance.

- The mathematical background which has been developed by the theorists (both semantic and algorithmic) allow better formal reasoning about parallel hardware.

The ILLIAC IV computer of the University of Illinois [Slo67] was well before its time especially in the first two of these factors. The hardware was too expensive for a complete configuration ever to be built and even that which was constructed suffered from unreliability. At the same time faster computers could be built by improving other parts of the system.

There is no doubt that these factors have eased to the extent of allowing huge developments in the field of parallel hardware and it seems likely that the next few years will see the development of even more concurrent architectures.

1.2.3 The pitfalls of parallelism

The pitfalls of parallelism are really just the limitations which parallel implementations have. However they are so fundamental that it is always worthwhile to point them out.

Shore. Having designed the *Associative Processor* [Sho73], an early logic-in-memory processor, Shore concluded that it was inappropriate for its design use (signal processing) because the *processing ratio* - the ratio of devices used as memory to those used for processing logic - was too high. He concludes that for systems with a high processing ratio, the chief aim of the software must be to keep the proportion of devices which are meaningfully active as high as possible.

In the light of current developments in VLSI it is dubious whether his conclusion is still valid. It can, however, be extended to apply to parallel systems where the equivalent aim is to keep as many processors as possible active. If, for instance, at any stage in the computation all the processors are waiting for one processor to complete its current task, the processing slows down to the speed of this one processor *bottleneck.*

Amdahl's Law This states [Amd67] that there is a proportion of processing which is inherently sequential. Amdahl terms this *data management housekeeping* and states that this takes up about 40% of the time, allowing that it could be reduced to 20%.

Assume that the housekeeping overhead is 20% and that this may be performed on a totally separate processor. Further assume that enough technological development has taken place to speed up the processing of the house-keeping

by a multiple of two. If the rest of the processing is totally parallelised to take effectively zero time, the total improvement can only be by factor of 10. This is obviously a useful speed-up, but is less than might have been expected.

It is only fair to point out that since these two papers were published many limitations which were then apparent have become less restrictive. In the former case, cost of the basic devices which VLSI has brought weakens the argument. In the latter case, many of the slow computations which these systems are designed to speed up have a small proportion of data management housekeeping and so are more amenable to improvement through the introduction of concurrency.

In both cases the restrictions *have not* been removed, they remain and have only been 'moved back'. The upshot is that these pitfalls remain and are just as deep. Although the immediate possibility of falling into them has been removed, they still remain to trap the over-ambitious and under-wary. For instance Amdahl's law must still hold (see for instance [DLS86, page 48] and several places in the following chapter). Indeed it is still the most critical software engineering issue in program parallelism. Fortunately several of those problems which are targets for being speeded up by parallelism are compute bound to the extent that the data management housekeeping may take up only a very small (say 1% or 2%) of the processing load. These problems are still subject to Amdahl's law and thus a basic speed up of the order of 50 to 100 times might be expected.

1.3 Pattern Generation

Chapter two describes what is involved in performing pattern generation, together with an explanation and justification of each of the stages. What follows is a description of where and when pattern generation occurs in the process of designing VLSI integrated circuits. Once aware of this, it should be possible to

define the limitations imposed upon the processing and so infer which level or levels of the implementation can be utilised to improve the performance.

Pattern generation is the final step of data processing which occurs before the physical processing associated with fabrication starts. The output of any integrated circuit design system, be it silicon compiler, gate-array personalisation system or full custom engineers work-station is a representation of the geometry which will exist on the final product (the chip). [1]

The first physical representation of this geometry is the *mask* which is used much as a photographic negative is, to project an image of the geometry onto the chip. *Mask making machines* convert geometric data presented in in the form of a *pattern tape* into physical masks. Their precise method of functioning is described in the next chapter.

For reasons which are fully explained in the next chapter, a considerable amount of processing needs to take place to convert the artwork representation into the pattern tape. This processing is termed pattern generation.

Data preparation for pattern generation is not dissimilar to compilation in that the (normally binary) output must be equivalent to the (often textual) input. Both input and output must be in a fixed form. Pattern generation suffers in a similar manner to compilation in that it incorporates functions which, although difficult to perform, appear easy to humans. In contrast to compilation, in pattern generation most of the high level structures of the input are not directly translatable to any structure in the input. Furthermore it is the norm for pattern generation input to be massive and pattern generation has no techniques which correspond to separate compilation.

---

[1] In fact pattern generation is not limited to integrated circuit pre-processing. A good pattern generation system should be able to handle *any* sort of pattern data - be it for electronic systems (when the product is integrated circuits), optical systems (diffraction gratings) or acoustic systems (surface acoustic wave guides).

1.3.1 The motivation for faster pattern generation

At first inspection, pattern generation is not an obvious candidate for speed enhancement. In sharp distinction to simulation and design rule checking, the traditionally time-consuming parts of the design process, pattern generation should only occur once. In fact it is not unknown for a design to go through many iterations even after the design has been sent to the mask shop.

The major motivation in speeding up pattern generation is not unrelated to Amdahl's argument. Considerable efforts have been and currently are being expended to speed up the design of integrated circuits. Leading the development are the fast prototyping services which will use the direct write devices mentioned in the next chapter. Once one bottleneck is removed it is bound to be replaced by another in a different part. Similarly once various parts of a design process have been speeded up the pattern generation stage will become the bottle-neck. Currently a large design may take several days to process prior to its manufacture as a mask. This is less than the turnaround offered by fast prototyping services. It is obviously essential to reduce the PG processing time so that it ceases to dominate the fabrication and preparation time.

1.3.2 A method of speeding up pattern generation

Pattern generation, then, is a part of the VLSI design process which requires speeding up. In many other parts of VLSI design systems great speedups can be made by taking advantage of the hierarchical nature of the input data (for instance [Whi81]). This has paid great bonuses in very many cases, as well a serving as a good way to control the complexity of the system during its design. Although it may prove possible to make very limited use of hierarchy in future pattern generation systems, dependency upon hierarchy in order to speed up processing is inappropriate for two main reasons:

1. Pattern generation systems need to be able to deal with totally 'flat' input, that is data which has had all the hierarchy removed. Quite often this is

the result of a previous attempt at pattern generation, the original data no longer being available.

2. Even were input restricted to hierarchical data, the concept of hierarchy is as varied as design styles. Because of this only the most general assumptions about the form of hierarchy present could be made. This in turn makes the processing considerably more difficult than it needs to be [BOW83].

Thus the chosen pattern generation system needs to be able to work upon totally flat data. It seems expedient to develop special purpose hardware to perform the process of pattern generation. In view of the amount of research currently being carried out in the field, a 'stage two' approach was decided upon. There are of course many alternative methods of speed improvement including the design of a special purpose piece of hardware. This would involve a higher cost than parallel machines (as well as being outwith the training of the author). In addition a parallel machine might prove useful in other regions of the design system. Given the plethora of different machines available it seemed pointless to develop a system targeted at any one hardware system. Rather it was decided to try to develop *techniques* which would allow pattern generation systems to run fast on a range of parallel MIMD hardware platforms. It was hoped that the applicability of such techniques might prove to be wider than just pattern generation processing.

Thus, in default of a single target architecture, the research and implementation were carried out in terms of broad restrictions which should encompass the greatest possible number of target architectures.

1.3.3 Restrictions upon pattern generation processing

The first stage in doing any development of any techniques must be finding the limitations and factors which control development of pattern generation

systems. Having found them the overall implementation techniques could be broadly formulated.

A fundamental restraint upon PG systems is the possible size of input data. This is where pattern generation sharply diverges from the closely related field of graphical image processing. In image processing it is possible to consider both pixel parallel (shape serial) and shape parallel (pixel serial) processing [Kil85] since the resultant image is considerably smaller than that involved in image generation - A graphical image is unlikely to be larger than 4k by 4k pixels or have no more than a few thousand objects to be shown. The equivalent representations of a state of the art design would be $>$ 40k pixels and have millions of objects. Of course a graphical image needs to be processed considerably more quickly than an integrated circuit.

Accuracy is critical to pattern generation systems in that the output should be as precise an image of the input as possible. All rounding (which needs to be done since mask making machines have fixed accuracy which is often much less than the input accuracy) should be performed with the greatest possible care so as to minimise inaccuracy. Internal rounding (rounding which occurs as a result of fixed internal resolution) should be avoided wherever possible.

After accuracy, speed is the most important factor, hence the decision to conduct this research into speeding up the pattern generation process. As previously noted the major constraint on the work in this thesis was that any developed software should be able to run on a wide range of all the existing parallel hardware. This in turn meant that unbounded parallelism could not be assumed since it might not be present in every case, although it is desirable that such parallelism which is present be used fully. Having specified that the amount of available parallelism is limited, it is important to ensure that all individual processors are kept busy.

Given that final choice of machine has been circumvented, it would be precipitate to assume a high communication bandwidth. Furthermore no particular topology for interconnect could be assumed and so any assumed topology must

be easily and cheaply implementable. This in turn means that communication must be considered as a restriction.

Finally, the processing units of most concurrent machines do not have unlimited or virtual memory. Any algorithms which run on concurrent hardware should not consume large amounts of memory or at worst they should have controllable memory requirement. Since memory is a cheap resource and (within limits) moderately easy to expand, the actual amount of memory available per processor can be considered to be large (but fixed). Of course, this limitation is not a restriction solely of multiprocessors since several uni-processors have a upper limit of 16 megabytes of user addressable memory.

To summarise, the five major restrictions are that any resulting system should be: accurate as described above, fast, making the best possible use of the available processing power, not overloading any communication system while only requiring a fixed amount of memory in each processing unit.

1.3.4 The Overall techniques used

The general technique used throughout the the generation of the parallel system was to take, where appropriate, an existing sequential system and modify it, substantially in some cases, less so in others, to run on parallel hardware. Since there was possibility of limited amounts of concurrency, fairly course grain parallelism was used throughout. The individual techniques are discussed in the relevant chapters. In each chapter mention is made of the guidance which each limitation gave towards the further development - the solution which serves to meet one limitation goes a long way towards meeting several others.

Speed It is the central supposition of this thesis that any resulting system will be faster than any existing system. By making the best possible use of available processors, the best possible speed up should be achievable. How much performance improvement might be expected for a given increase in hardware size is discussed below.

Accuracy The basic system which served as the foundation for the research is acceptably accurate. It remains to ensure that the additions to the system do not introduce further errors. The fact that the resulting system was a direct derivation of an existing one means that any changes made to the original system in order to enhance accuracy could immediately be incorporated into the parallel system. This of course goes for all the limitations which are no *per se* limits introduced by parallelism (for instance memory requirement limitation).

Communication As noted above communication is a source of bottle-necks, the importance of the bottle-neck increasing as both the amount of concurrency and the speed of the basic processing units is increased. A good general rule to observe throughout is to reduce to a minimum, and if possible remove, any interprocessor communication. This leaves the sole communication being the distribution of the input data to the processing units and the transmission of the physical output data onto the necessary output media. Furthermore it was decided that whenever there was a choice in pay-off between memory, speed and causing extra communication the first two would be favoured over the latter.

Load balancing This requires that each of the processing units is given the same workload and so will complete at approximately the same time. This in turn requires that some relationship between some measure, applied to the input data, and processing time is established. Having found this measure and the associated relationship, some method must be found to divide up the processing effort equally. The measurement of the input data, the corresponding processing load analysis and data division should be easy to perform or the process of balancing the load will become so time consuming as to be self defeating.

Controlling memory use takes on two forms. Firstly it requires establishing whether the data which is memory resident *needs* to be resident for each processor or whether a different processing philosophy could allow the data to be processed in such a manner that no processor will ever exceed its memory

capability. If this cannot be done an assessment of memory requirement, preferably using the same input measure as used to gauge processing time, can then ensure that no processor ever 'runs out' of memory.

If the method of division and the ways of measurement are kept cheap, in terms of required processing, it is not unreasonable to assume that the speed up achieved is of the same order of magnitude as the increase in complexity of the hardware.

1.4 This Thesis

Having chosen a system in need of temporal performance enhancement, defined the limitations and developed the ground rules for implementing the parallelism, the actual business of speeding up the individual programs could commence. In doing this, the techniques themselves could be developed.

During the development of the parallel system, analysis needed to be carried out to ensure that the actual system would be faster. Other characteristics of the system which needed to be measured were the memory requirements and the communication bandwidth.

In each case the analysis and measurement of the developed system was by an emulation of the parallel system. Given such compiler technology and operating system support for communication as is needed, it would not be hard to make the system run on any real parallel processor.

The precise form which each emulation took is described in the relevant chapter. In general the concurrency was emulated by careful use of timing procedures placed around an iterative loop which simulated each processor in turn. Timing information was taken by associating a 'clock' with each process being emulated. This clock was started and stopped as processing was performed on behalf of each process. Support routines provided such extra information as finding out which processor became free first. Bandwidth was measured by gen-

erating a time-stamp log which could be analysed later. Memory requirements were measured by using calls to functions inherent in the operating system of the emulation computer. Great care had to be taken when communication was emulated, especially when bandwidth measurements were also being taken.

It must be borne in mind that the resulting emulation is of a system with processors which have the same power as the processor which performed the emulation. The bandwidth measurements would of course differ if the processing units were of a different power. This is not seen as a difficulty since what is being presented is a method of achieving speed up by using parallelism not a single parallel system.

In this way a good measure of the performance of the parallel system could be achieved in that if the system was not well behaved according to any of the constraints then the results would show this. However if the system was well behaved, the results could be assumed to be a reflection of a reasonably well performed implementation upon real parallel architecture. For instance although communication was fast in the emulation, a similar rate could be achieved in an implementation by utilising such techniques as double buffering and asynchrony - the results of the bandwidth measurement being low enough to ensure this; any remaining overhead would be mirrored in the emulation as time taken to write the trace file.

1.4.1 Description of the rest of the thesis

The rest of this thesis falls into three parts. Chapter two describes the complete process or pattern generation (both the mask making and associated data preparation) in greater detail and gives some details of how one particular pattern generation system works. This is unfortunately required to motivate and support the next three chapters. Chapters three, four and five describe the actual work which was carried out. They are all of roughly the same form. Where it is appropriate a description of the range of algorithms possible and the chosen 'sequential' algorithm is given. There then follows a description of

the resultant, parallel algorithm, with justification for why it was chosen. Results, both the expected and (in a few cases) the unexpected are then given, and conclusions drawn. For each chapter important results are the maximum number of processors allowed for any stage in the system, this being governed by the input or output bottle-neck and secondly the data bandwidth which this number of processors require. Finally for each chapter there are several loose ends, most of which are the result of hindsight and are indications of the ways in which further research should take place.

The final chapter ties together the preceding chapters and draws some conclusions. There is a description of possible further developments incorporating other facets of implementation improvement. The implications of the developed techniques for applications other than pattern generation are discussed. The implications for architectures are also discussed.

Chapter 2

Data Preparation for Pattern Generation

2.1 Mask Making

The function of a mask making machine is to convert a digital representation of the design, the pattern tape, into a physical representation, the mask set. The masks, which are normally copies of a *master set*, are used during the actual fabrication of the integrated circuit wafer.

Integrated circuit fabrication is a multi-stage operation. At the start of each stage a two dimensional pattern needs to be made upon the wafer surface. This pattern will control whether or not parts of the wafer are exposed to various processes - for instance ion implantation, ion diffusion, or metal or polysilicon deposition. The patterning is achieved by covering the entire wafer with a layer of *photo-resist*. The mask is brought up to the wafer surface and a light source (visible or u.v.) exposes the wafer. The light affects the resist, either breaking it down to make it soluble, or making it insoluble. In either case the excess resist is removed leaving the desired pattern upon the wafer surface to allow further processing. Integrated circuit design is the function of designing the precise geometry which is desired upon the surface of the wafer and thus the completed chips. It is the job of the mask making machine to generate the mask sets from a suitable representation of this data.

The earliest mask making systems were purely manual. Each layer was check-plotted at a large magnification. The precise image of the design was

hand cut and peeled into rubylith. The rubylith was photo-reduced to generate a *reticle* (normally a ten times copy) which was further reduced and duplicated in a *step and repeat machine* to the size required for the master mask.

The hand cutting and peeling of rubylith was very tedious, error prone and time consuming even at small scales of integration. Thus methods of automating the mask making process were introduced. The earliest development was the automation of the cutting process. The rubylith was cut by a device similar (and occasionally identical) to a flatbed plotter. The peeling stage continued to be performed manually - this was still time consuming and even more tedious than before. The way was therefore laid for the development of mask making machines which required very little operator intervention.

Modern mask making machines can be divided into two major types. The earlier, but still popular, photomechanical machines [Hen77], and the more recent electron beam machines. Photomechanical mask makers tend to be less accurate (in the sense of working to lesser tolerances) than electron beam machines; however the additional accuracy of electron beam machines is offset by the reduced cost, both capital and working, of photomechanically generated masks as well as the lessened problems attendant when working with a mature technology. Thus photomechanical mask making machines, although decreasing in popularity are still in the majority [Tra85]. Indeed current developments indicate that photomask making machines will exhibit a longevity which even its proponents would not have originally envisaged.

2.1.1 Photo-mechanical mask makers

Photo-mechanical (or *optical*) mask makers are those which are usually described in text books on VLSI design [MC80, section 4.2] [HS80]. A transparent *plate* (made usually of glass or quartz) is coated with a photographic emulsion. This plate is mounted onto a table which in turn is sighted below a movable light projection system with a rectangular aperture whose size may be varied. The plate can be moved relative to the light source along the X, Y and rota-

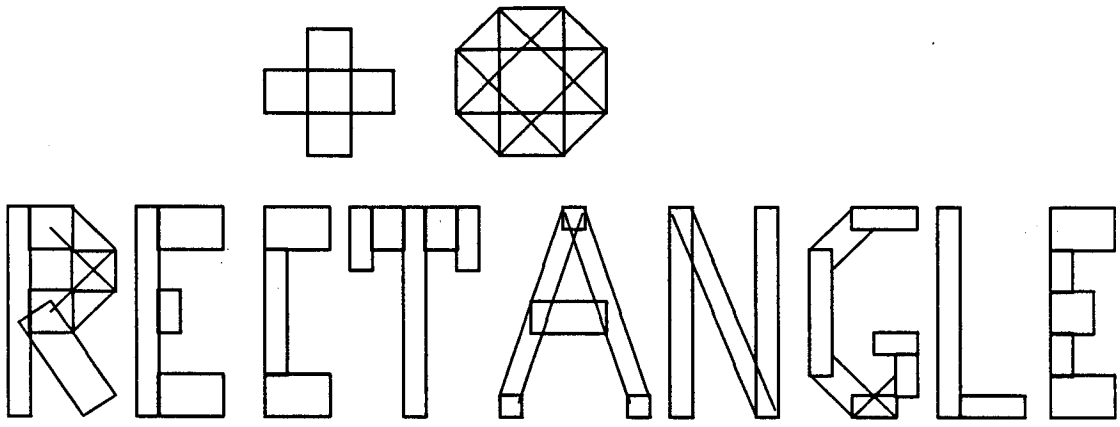


Figure 2–1: Optical P.G. Data

tional axes. In addition both the length and the width of the aperture can be adjusted. These are the five main parameters to the mask making machine; there are others, but these have no impact upon the related data preparation.

The pattern tape for the generation of masks for an optical machine consists of a series of quintuples, controlling the five major parameters. The mask maker functions by reading a tuple, adjusting its settings and then *flashing* an exposure. Figure 2–1 illustrates typical optical pattern generation data. Once the complete tape has been read, the plate is removed and developed in a similar manner to a photographic negative.

Typically the precision of optical mask makers is less than that required for the final mask[1]. Therefore the plate is often exposed at a ten time scale generating a 10× *reticle* which needs to be *step and repeated* to generate the master mask. When generating masks from reticles it is occasionally possible to combine two reticles thus allowing two patterns on the final mask. A combination

---

[1]Typical parameters are: smallest box size (length and width) 4μ, box size increment 0.5μ, positional increment 0.1μ. These figures are those which are accepted by the machine. It is unclear whether they relate to the accuracy or the tolerance of the machine.

of more than two reticles is usually inadvisable due to realignment accuracy problems with the step and repeat machine. As the tolerance and accuracy of newer generations of optical mask makers get better, a trend is developing to generate 1× *Reticles* which require no photoreduction.

Side effects of optical mask-making

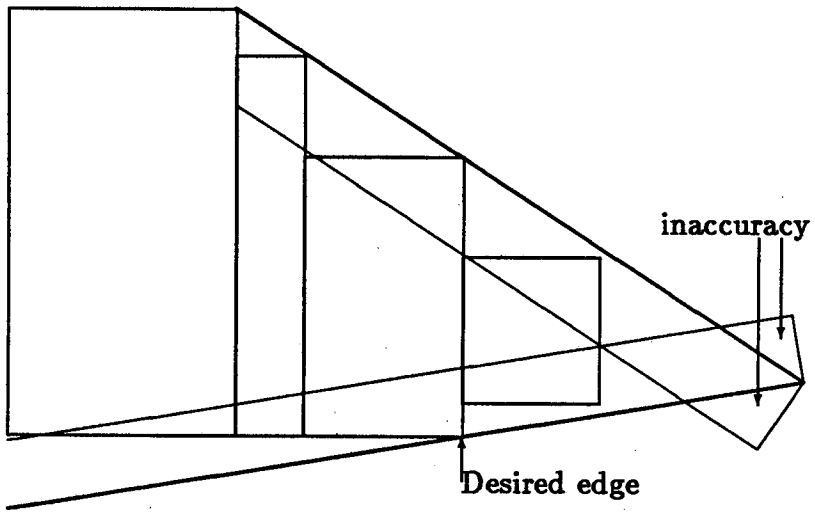


Figure 2–2: Acute angle approximation for optical P.G.

Because optical mask makers can only generate geometry which is made up of rectangles, not every input shape can be generated. In particular geometry containing acute angles cannot be represented and an element of approximation needs to be introduced - see Figure 2–2.

More problematic are *flaring* and *fogging*. It can be assumed that the first time an area of emulsion is exposed a rectangle corresponding to the input parameters is generated on the plate. Subsequent (over)exposures of the same area cause sideways scatter and thus exposure of areas *not* below the flash. The resulting inaccuracy manifests itself into two ways - flaring and fogging.

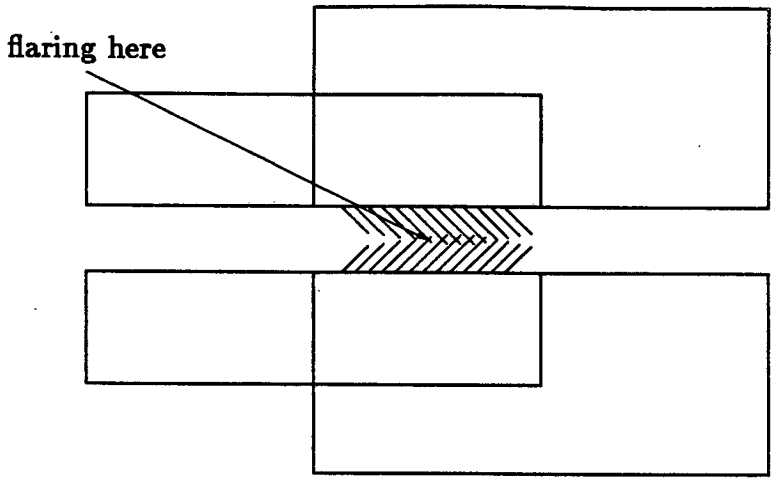


Figure 2–3: Flaring

Flaring When an edge of (for instance) a wire is flashed multiply, the sideways scatter causes flaring - see Figure 2–3. It is easy to see that this could cause design rule violations and subsequent reduction in yield in the final chips.

Fogging If an area is multiply overexposed scatter into neighbouring non-exposed areas can cause the fogging (partial exposure) of these regions - even if there has been no overlap along any edge. The amount of over-exposure has to be large (the order of tens of flashes not ones). There are cases where overlap can be beneficial since it can reduce flash count.

The speed which an optical mask maker takes to expose a plate is obviously a function of the input size. However the time taken to alter parameters also contributes sizeably. Well ordered data can be flashed at an average rate of 150 flashes per second. Flashing badly ordered data can be as slow as 2 flashes per second.

2.1.2 Electron Beam mask-makers

In electron beam (or *Ebeam*) mask making the plate is exposed by a beam of electrons which may be deflected across the plate. As for graphics displays

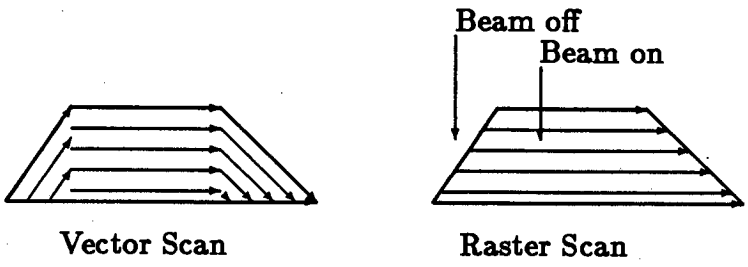


Figure 2–4: Raster and Vector Scan Lithography

which are, to an extent, based upon a similar technology, Ebeam machines may be divided up into up into vector scan and raster scan devices.

Raster Scan Devices are the more numerous type of device. As the pattern tape is read the pattern to be exposed upon the plate is written into a bit-map memory. This is scanned a line at a time; at the same time the electron beam scans across the surface of the design. The pattern in the bit-map controls whether the electron beam is turned on or not and thus where the emulsion is exposed.

Vector Scan Devices do not use a bit-map memory. As each shape is read from the pattern tape it is converted into a series of *rasters* which the electron beam then describes upon the surface of the plate. The earliest machines were of this type and the very latest machines have this capability. As well as the traditional problems for pattern generation outlined below, vector scan Ebeam machines present the pattern generation systems with the additional problems of dosage calculation due to proximity effect [Cha75] (an effect not dissimilar but unrelated from flaring). The calculations involved are somewhat complex and are not addressed in this thesis.

Figure 2–4 shows how the two types of Ebeam machine would generate a simple shape. Despite the difference in operation, the input formats, both in terms of overall organisation of the data and in terms of the individual data items, for the two types of machine are surprisingly similar. Neither vector scan

nor raster scan machines can expose a complete plate in one operation; rather the area of the plate is divided up into regions termed *stripes* or *tiles* which have to be processed in a fixed order. In raster scan machines the stripes, which are normally wide but short[2] have to be processed starting at the bottom left in increasing y value to form a *segment.* The segments are processed from left to right. In vector scan machines the regions are square[3] and processed in boustrophedonic (serpentine) order.

The pattern tape for Ebeam machines consists of a series of subfiles, one for each stripe. The subfiles have to be organised on the tape in the order in which the stripes will be processed. Each subfile consists of a series of descriptions of trapezia (quadrilaterals with two parallel sides) with the parallel sides being horizontal. Squares and triangles are special cases of trapezia. Figure 2–5 illustrates a typical Ebeam output. Trapezia are chosen since:

- All shapes may be represented as collections of trapezia.
- Trapezia may be easily vectorised.
- They are easily converted into bitmap format (rasterised)

Side effects of electron beam mask makers

The methods of functioning both of raster and vector scan machines is such that all data may be represented (to the level of accuracy of the machine) without the need for approximation which was required for optical machines. Although the data is still rounded to the nearest address unit, the levels of

---

[2]Typically 32768 by 512 or 1024 address units. The *address unit* (a.u.) is the resolution of machine. Typically it is a few tenths of a micron

[3]Typically 32768 a.u each side

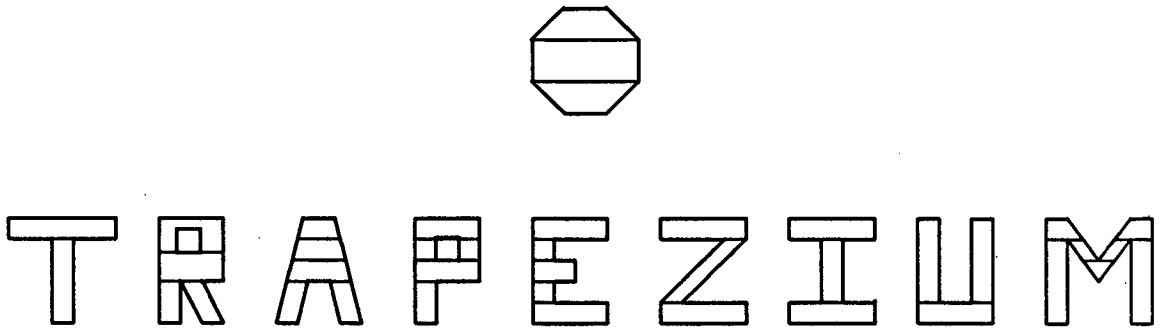


Figure 2–5: EBeam P.G. Data

representational accuracy achievable by Ebeam machines are still much greater than those offered by optical systems.

Since raster scan machines represent the pattern in a bitmap there is no side effect of overlapping. However vector scan machines still suffer with sideways scatter with the attendant problems of fogging and flaring when an area is multiply exposed.

2.1.3 Other Pattern Generation Equipment

Although mask making equipment accounts for the vast majority of pattern generation systems there are other types of equipment which fall within the bounds of pattern generation and should, for the sake of completeness, be mentioned.

Direct Write and related devices

Machines of this class circumvent the requirement for masks by actually working on the wafer surface. The vast majority of devices in this class are still at the prototype stage. The motivation behind the development of these systems is to enhance the speed of processing of designs. It seems feasible to assume that direct write on wafer machines will be able to process ten layers in one hour.

This obviously increases the need for faster data preparation. There are two main classes, direct write systems and deposition systems.

Direct Write on Wafer Devices are used in the patterning for any layer during processing. Direct write devices function by using an electron beam to remove resist without the intermediate stages of mask making, exposure and development.

Laser and Plasma Deposition systems obviate the need for resist completely by depositing the upper (metal) layers of a semicustom design directly. In addition some systems can etch away layers (for instance for vias).

Formats and side effects. Systems of both these types are so new that their input formats are not known. However it seems likely that they will be similar to those for Ebeam machines (input based on simple shapes, covering discrete regions). Similarly the restrictions and side effects of these systems are unknown although avoidance of multiple exposure is vital for deposition and direct write systems.

Mask Checking Machines

The purpose of a mask checker is to ensure that there has been no error in the mask making process and that the masks reflect the input data. Early mask checkers functioned optically, more recent systems work electronically. As far as data preparation is concerned these are very similar to Ebeam machines, indeed in some cases the same data will drive both the mask maker and the mask checker.

Laser Driven Optical Equipment

Another recent development is the raster scan, laser driven optical mask maker [SR87]. This functions in a similar manner to a raster scan Ebeam machine,

but uses light (in the form of a laser) rather than electrons to expose the mask. The similarity of function is mirrored by the fact that these machines take as input format the 'industry standard' formats which are used to drive the Ebeam machines. Thus as far as data preparation is concerned these may be viewed as equivalent to Ebeam machines.

2.2 The Function of Data Preparation for P.G

The ultimate purpose of pattern generation data preparation is to generate a pattern tape. The input is usually data expressed in an *artwork format* although *P.G. format* input is not unknown. P.G. data is that which forms the input to a mask maker[4]. Artwork data is the result of a design system. P.G. formats are rather more restricted than artwork formats in the following ways:

- As discussed earlier the shapes representable by a mask maker, and thus by a P.G. format, are limited. Artwork formats normally allow general polygons, boxes and wires (the last two being special cases of polygons).

- Artwork formats allow hierarchy. P.G. formats are completely flat.

- In P.G. formats the layers which will make up the individual masks need to be specified in order in their entirety. Artwork formats allow shapes to be specified on any layer at any time.

- Data which is an artefact of the design system and which will not cause geometry to appear on the masks. Examples include nodes, stretch points and text.

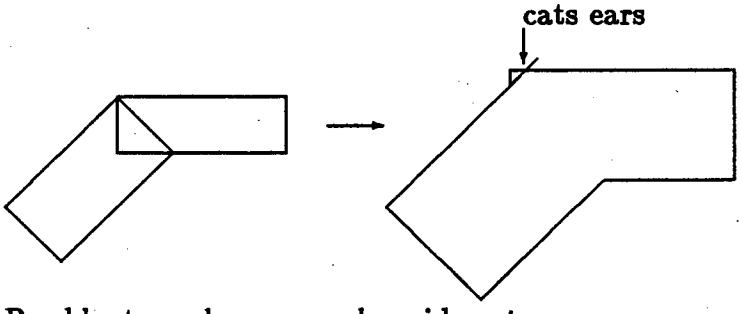
---

[4]The term 'P.G. data' is often taken to mean data for optical machines only. In this thesis it refers to data for *all* types of pattern generation equipment discussed above.

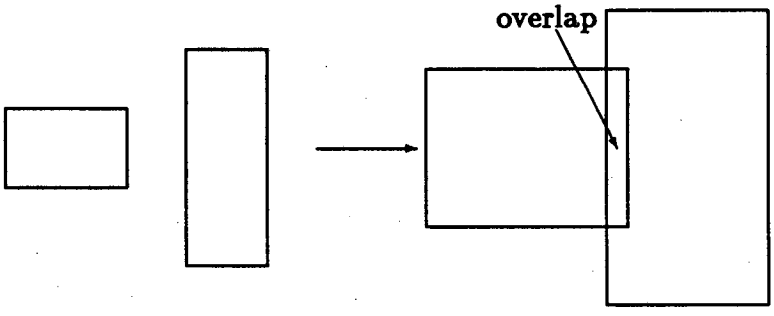
To summarise, artwork formats reflect the ways that design systems and designers think and work, P.G. formats reflect the way in which mask makers function.

Pattern generation data preparation must perform the conversion from an artwork format to a P.G. format. In doing so it must take into account the side effects of the mask maker. In addition it may need to perform other operations which are the result of side effects of both the design style and the fabrication process. Functions which it should fulfil over and above the conversion include:

Overlap Removal. This ensures that there is no fogging or flaring during the mask making process. If the overlap removal is achieved by joining together all overlapping *and* abutting shapes into polygons it makes the sizing considerably simpler.



Pre-bloat overlap removal avoids *cats ears*



Overlap removal needed after bloating

Figure 2–6: Bloating and overlap removal

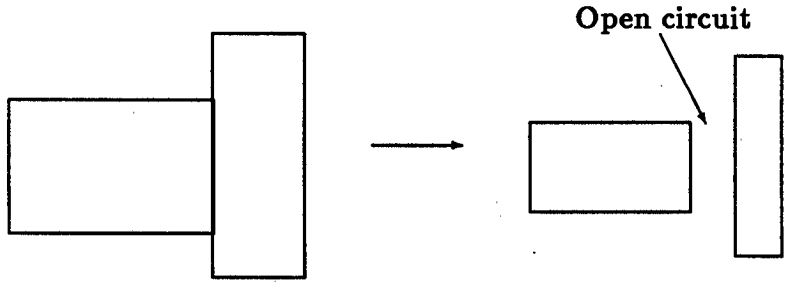
Sizing. Very often the fabrication techniques are such that the pattern made upon the plate should be larger or smaller than specified by the designer. This *sizing* is achieved by moving each edge by an *absolute* value in a normal direction. Thus the pattern (or field) becomes larger (respectively smaller) in that it covers a greater (respectively lesser) proportion of the area of the design. Performing of sizing is the most important function of P.G. data preparation systems. Both positive sizing (*bloating*) and negative sizing (*shrinking*) need to be carried out in conjunction with overlap removal. In fact as Figures 2–6 and 2–7 show, overlap removal needs to be performed both before and after sizing. When sizing angles, the apex will move by more than the amount of sizing. It is often useful to clip angles to reduce this effect. See for instance [BB80] or [Kil82].

Mask Generation. Often the precise masks required do not correspond one to one with the layers specified by the designer or the design systems. For instance, the implant region for an nMOS process may be generated from the *union* of active area for depletion mode transistors and buried contacts, both suitably sized (possibly by different amounts). Other functions which are needed are *intersection* and *subtraction*. Simple Boolean algebra ensures that given these three functions (which can correspond to the Boolean operations of *or*, *and* and *not*) any function can be generated. However, in addition to these functions it is useful also to have an explicit (rather than derived) *exclusive or* function.

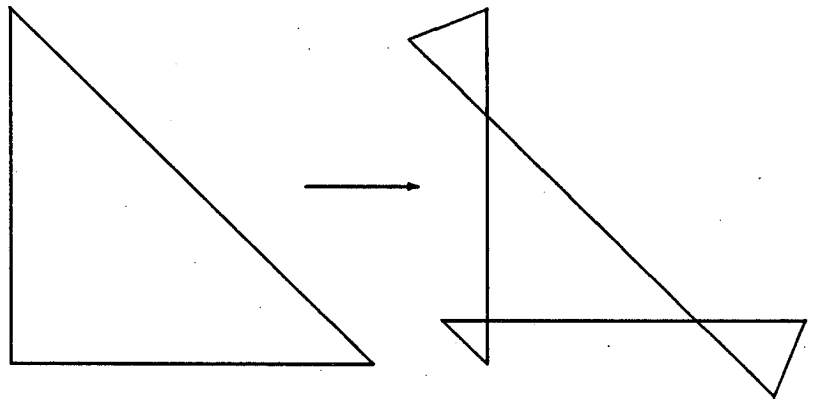
Pattern inversion. Depending upon the polarity of the resist used in fabrication it is occasionally necessary to invert the pattern - that is to make areas of pattern into holes and holes into patterns. This is usually achieved by subtracting the design from a box whose size is the design's bounding box.

Scaling. This trivially performed operation is vital, not only when generating pattern tape for 10× reticles, but also when reducing existing designs. As fab-line tolerances improve, it is quite common to reduce mask sizes by

reprocessing a design with a scale of 0.9 or 0.8. Scaling is achieved by moving each edge by a *relative* amount. The proportion of the area which is occupied by the pattern in a scaled design is the same as the original. The overall area is different. Scaling is performed by multiplying the coordinates of each vertex by the scaling factor.



Shrinking could cause open circuits



Shrink causes badly formed polygons

Figure 2–7: Shrinking and overlap removal

2.3 A P.G. Data Preparation System

Most PG systems function in two stages. The first stage involves performing most of the functions described above. The input data is parsed and converted into a totally flat internal format. This internal format is chosen so as to be suitable for the implementation of sizing, scaling and Boolean operations. These operations are applied repetitively until the geometry representing the final masks is achieved.

The second stage of the processing involves the conversion of this internal format into a suitable format for presentation to the mask maker. According to the precise internal format used, this may be purely a translation or involve a conversion of form as well as format. For instance, if the intermediate format is based upon polygons (lists of vertices), the second stage needs to involve conversion of form from edge-based representation to area-based representation. However if the internal format was based upon trapezia, conversion to Ebeam output becomes trivial. Finally the programs which make up the second stage need to sort the PG data into the appropriate order.

Since there may be more than one form of output, the second stage is likely to consist of more than one program. Thus it is sensible to migrate as much function as possible into the first stage.

The system which forms the implementational basis of this research is illustrated in Figure 2–8. The first (*merge*) stage carries out the majority of the pattern generation function on data represented as *polygons*. The second stage then has to decompose (*fracture*) these (arbitrarily complex) polygons into the simple shapes which make up PG formats and apply suitable ordering to them. Thus there are three separate programs whose analysis and conversion to run in parallel form the basis of this thesis: merge, Ebeam Fracture and Optical Fracture. These make up the next three chapters.

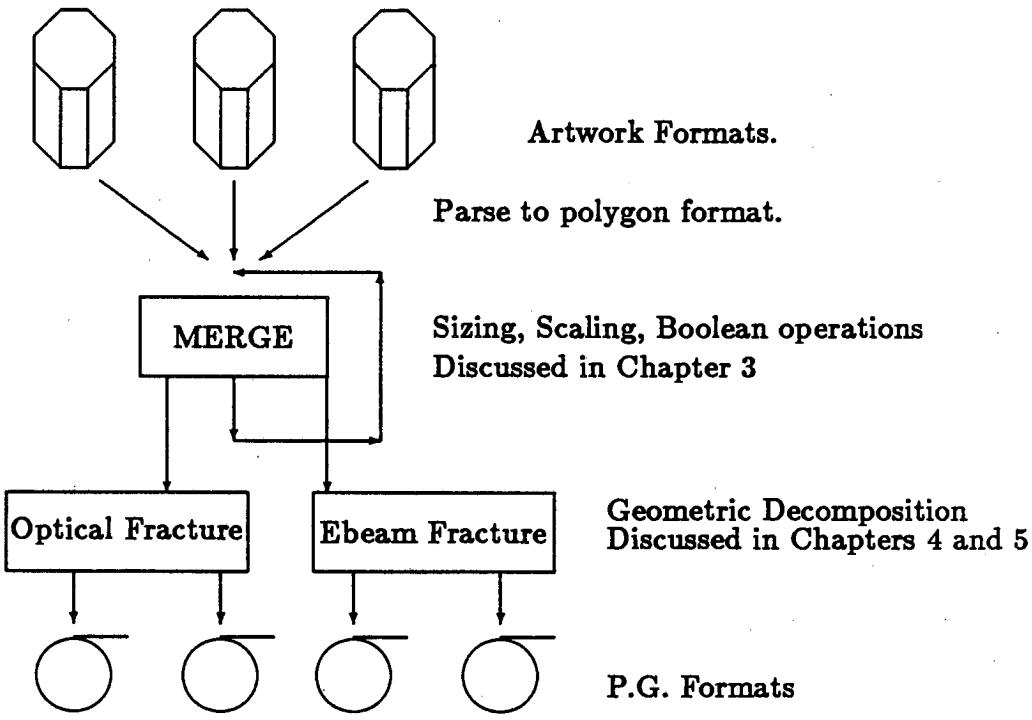


Figure 2–8: Overall Architecture of a P.G. System

Chapter 3

The Polygon Merge Stage

The first stage in the generation of mask-making data is the 'merge' stage. This performs the functions of:

- Sizing
- Overlap removal
- Other Boolean operations

3.1 Algorithms for Merging

When examining algorithms for use in the merge stage of a P.G. system there are several important factors to note. As mentioned in Chapter 1 it is highly important that the performance is as near optimal as possible, both in the best and in the worst case. At the theoretical level it is advantageous if the system performance degrades gracefully as the input becomes 'harder to process', that is to say as it moves from being the best to the worst case. At the software engineering level the algorithm should take advantage of known properties of the input data (if there are any), such as Guting's algorithms for merge of C-oriented polygons [Gut84] or Bentley et al's algorithms for (Manhattan) design rule checking, based upon statistical analysis of IC data [BHH80].

3.1.1 Theoretical Preamble

In [Sha78, Chapter 5] Shamos investigates the *intersection* of shapes - both the *reporting* and the *counting* of the intersections. We shall look at the algorithms that he describes and their descendants (which include the one used in this research) later. At this point it will suffice to state his key results.

1. *Intersection generation* (ie the description of the intersection or union) of two convex polygons may be carried out in time linearly proportional to the number of vertices involved. This result can be extended to monotonic polygons.

2. *Intersection detection* of arbitrary polygons can be done in $O(n \log n)$ time.

3. *Intersection generation* of arbitrary polygons has a worst case run-time of $O(n^2)$.

This last result is because arbitrary polygons may intersect in such a manner as to yield $O(n^2)$ intersection points. Shamos cites 'chicken feet' but of greater concern in processing integrated circuit data is the Manhattan 'egg crate' case. See Figure 3–1.

Thus although Manhattan geometry may give simpler algorithms the asymptotic worst case performance is the same as for general geometry. In fact egg-crates do occur in IC design - usually as the gate of large transistors (see Figure 3–2) - although they rarely occur in sizes large enough for the quadratic time complexity to cause embarrassment. Note that all these bounds refer to the intersection of two polygons whereas merging is concerned with the intersection of many polygons. However, as always the simple case can give indications of possible methods of dealing with the complex.

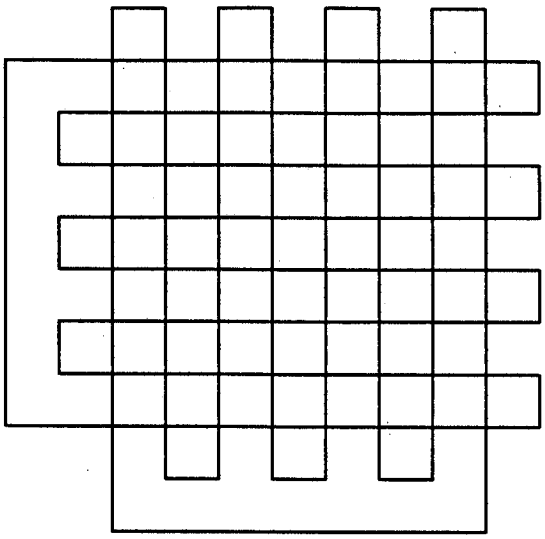


Figure 3–1: Egg crates

3.1.2 Area Based Algorithms

In this class of algorithm the data is represented by the area it covers, rather than the edges. The expression of Boolean operations is simple, but biasing becomes difficult.

Bit mapping

The most obvious method of performing merge operations is via *bit-mapping.* All the input shapes are rasterised into a region of memory.

By careful choice of the operations which are applied to the bit-map the union, intersection, difference and so forth are trivially calculated. These being especially useful in design rule checking. In [Wil80] Wilmore describes a development whereby memory utilisation is reduced by using a hierarchy (of memory) and effectively only changes are stored.

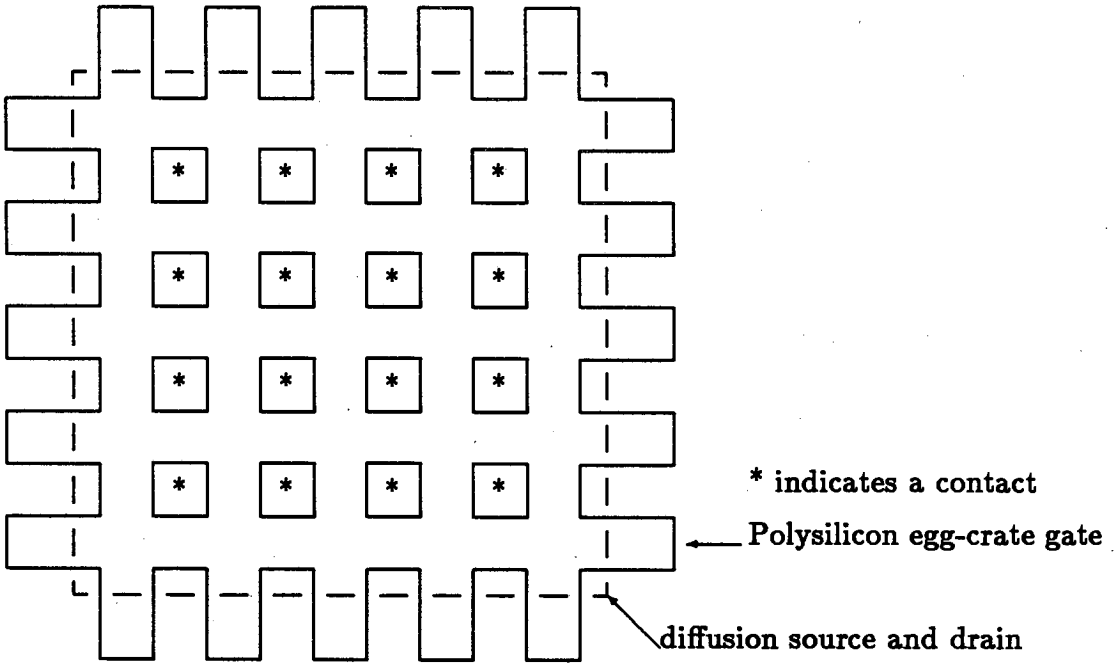


Figure 3–2: Egg crates in Large Gated transistors

As mentioned above biasing is not simple to implement. It could be achieved by some form of neighbour to neighbour negotiation between pixels. The representation is the same as that used by raster scan Ebeam machines. There would be obvious advantages if some method of exploiting this commonality could be found.

Bit mapping is an obvious choice for hardware acceleration (see for example [Sei82]) in that the basic operations are simple and are replicated over the entire design. Close analysis shows that however appealing this approach may be, bit mapping is not really applicable for pattern generation:

- Arbitrary angled geometry can only be represented by the same approximations which raster scan Ebeam machines use in the final stage before mask making (jagged edges). This representation is cumbersome and does not lend itself to the regeneration of other representations.

- The space requirement is massive. Even the processing of designs which are currently generated (**not** state of the art) would require hundreds of megabytes of bitmap. For special purpose hardware each bit needs to have associated logic thus adding to the cost. Obviously a certain amount of saving would be achieved by repeated use of a smaller processor array.

- Unless direct access is available to the bitmap store of raster Ebeam machines trapezoids need to be generated from the representation. Such access is not available. For optical P.G., rectangles (complete with the necessary approximation) need to be generated. In both cases it is not obvious how to achieve this 'de-rasterisation'.

Bit-map representations were considered not to be very useful as a basis for hardware accelerated pattern generation. It is necessary to investigate other ways of representation and manipulation.

Quad Trees

Quad trees (and their three dimensional counterpart Oct-Trees) [ABJN85] can be viewed as a variation of bit-map representations, in some ways related to Wilmore's hierarchical bit-map format. Each shape is represented by a tree. The root of the tree covers the bounding box of the entire shape. This area is recursively divided into four equal parts and these are represented by the four children of the subtree. The recursion continues until every leaf of the tree

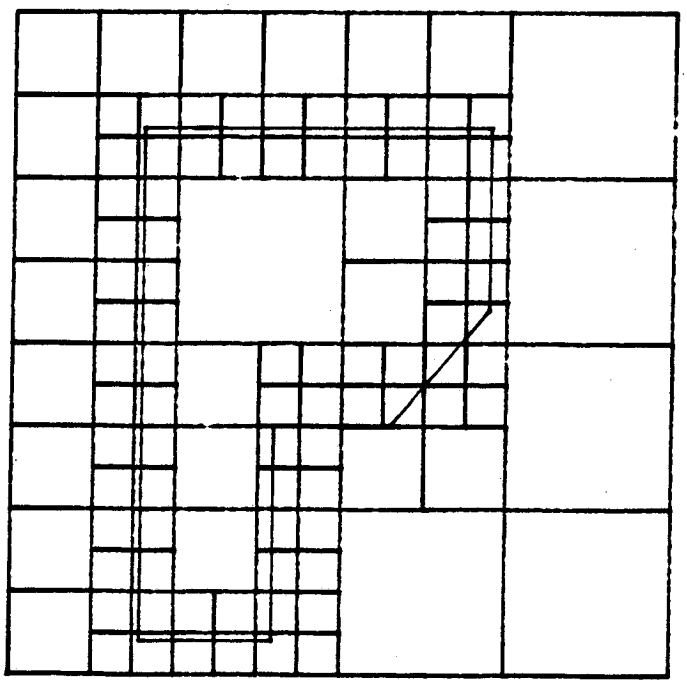


Figure 3–3: Quad Tree representation

represents an area which is either white or black or the level of resolution met. See Figure 3–3.

Shape intersection is fast but for pattern generation every shape would need to be intersected with every other shape - obviously a great inefficiency, although the 4-dimensional binary tree approach described below can reduce this. The essentially unbalanced and unbounded nature of the tree representation means

that any hardware would either be under-used or it would overflow. Furthermore the communication costs would be large.

3.1.3 Edge Based Algorithms

In edge based algorithms the shapes are represented, not by the area they cover, but by their boundaries. As Weiler points out in [Wei80], general polygon comparison can be carried out by simple operations on the *graph* of the boundaries of the polygons; he gives examples of the generation of union, intersection, difference and clipping given this graph. Although this generation is fast, involving one pass on the graph, the construction of the graph is difficult. Weiler only gives the trivial algorithm for generating it - namely the comparison of every edge with every other edge. Although this is, of course, worst case optimal, in most cases the $O(n^2)$ run time is totally unacceptable. All developments of edge based algorithms have been aimed at reducing the number of edge-edge comparisons where possible. To reduce the number of edge-edge comparisons methods need to be sought in which lines will only be compared where there is a possibility of them crossing ie a method should be found to *guarantee* that they do not cross. Apart from Scan line algorithms which are described later two methods of doing this have been proposed, Four Dimensional Binary Search Trees and Hierarchical Bounding-Boxes.

Four Dimensional Binary Search Trees

This technique, described in [Lau78], is used to reduce the number of polygon-polygon comparisons needed when performing design rule checking. Polygons are only compared when their bounding boxes overlap and the number of times that this check takes place is limited by using a binary 4 dimensional tree.

Each polygon is stored as a node of the tree. At each 'layer' of the tree one of 4 keys (comparison of the 2 extrema of the bounding boxes in X and Y) are used, in turn, to ascertain whether the next polygon is inserted to the left or

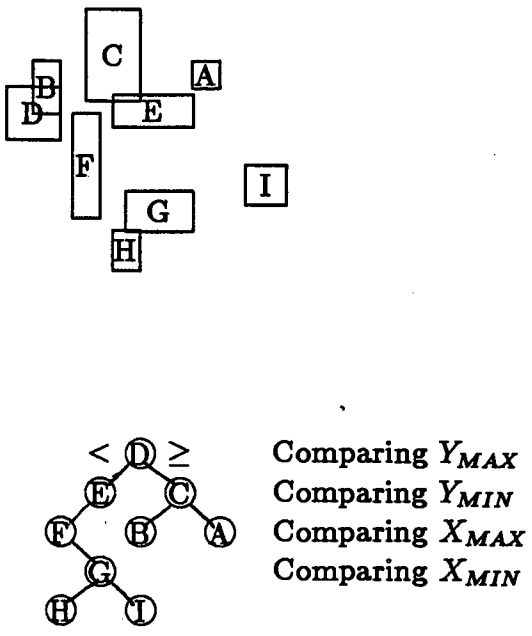


Figure 3–4: Four dimensional binary search trees

right of the sub-tree see Figure 3–4. By careful tree-traversal the number of polygon comparisons is limited to approximately $O(nh)$ where n is the number of polygons and h is the maximum height of the tree.

It is not obvious how this approach generalises to merging nor is it apparent how to build special purpose hardware. The tree generation would be simple to perform in hardware but the tree traversal would require considerable communications overhead which is contrary to the requirement of low communication costs.

Hierarchical Bounding Boxes

Hierarchical Bounding Boxes (or binary searchable polygon representations) [Bur77] are another method of speeding up polygon-polygon intersection. A tree is built - the leaves of which represent line segments which are monotonic in X and Y; these can be easily compared.

Associated with the line segment is a bounding box (a section rectangle). The section rectangles are recursively paired together until the root node, whose associated section rectangle is the bounding box of the complete polygon, is reached. See Figure 3–5.

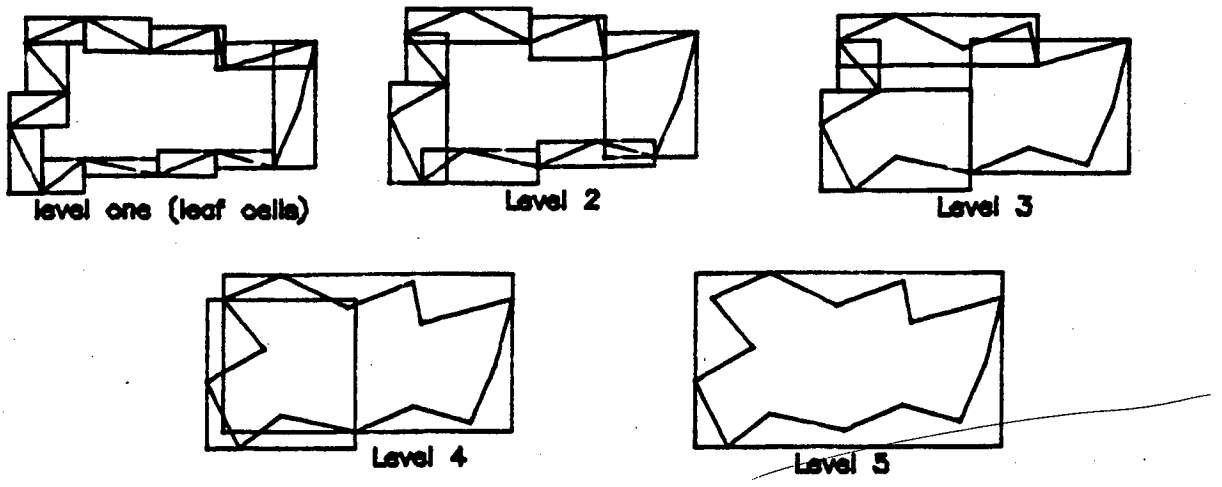


Figure 3–5: Hierarchical bounding box representation of a line segment

Searching for intersection between any two polygons is a matter of (successive) binary searches through the trees representing the shapes. As soon at the bounding boxes cease to overlap (a simple test to make) the search for intersections can stop. If the section rectangles of the leaf cells overlap, then the line segments can be (cheaply) compared.

The ease of representing complex shapes (for instance areas in maps) make this approach attractive at first glance, the more so since is can deal with curves - a great boon when bloating acute angles this being used to great effect in the Barton-Buchanan Polygon Package [BB80] . Unless 4-D binary trees are used the algorithm is still quadratic - each shape needs to be compared with each other shape. Furthermore the representation is at its best when representing predominantly convex shapes - outlines of counties for instance.

Pattern generation data with its propensity for wires (long thin shapes, as described in [BHH80]) does not lend itself well to this method.

3.1.4 Scan Line Algorithms

In [Sha78, Chapter 5] Shamos presents algorithms for the generation of the intersection of two convex polygons and for the detection of intersection of two arbitrary polygons. These algorithms are described below, since they are the forerunners of the class of algorithms known as *plane sweep* or *scan line* algorithms. It is interesting to study the development of these algorithms.

Intersection Generation of Convex Polygons

Each of the two polygons is divided into two separate 'chains' of vertices. The chains are a collection of continuous vertices delimited by the extrema of the polygons in X. The four chains are monotonic in X and may be combined in linear time. Vertical lines drawn through these points divide the plane into $n + m + 1$ 'slabs' (n and m being the vertex count of the two polygons) - See Figure 3–6. The intersection of any slab and either polygon is bound to be a trapezium (possibly degenerate). Intersection detection of trapezia takes constant time thus the intersection *detection* can be carried out in linear time.

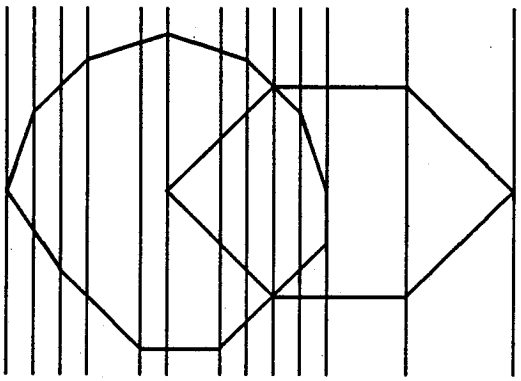


Figure 3–6: Dividing two convex polygons into Slabs

An imaginary scan line is 'swept' along the combined chain. At any point the slab defined by the line in its current position and its previous position will

contain at most two trapezia - one for each of the polygons. The intersection of these trapezia will have a maximum of six points only two of which can lie within the slab (as opposed to on its boundary). Thus the upper and lower chains of the resultant polygon is formed in one pass. These chains may be joined together in linear time. Thus the complete algorithm is linear. (see [Sha78, page 116-119])

It is simple to see how this algorithm may be adapted to generate the union of the polygons. Furthermore it is easily extended to work with convex line segments and is equally suited for use with monotonic polygons.

Intersection Detection of Arbitrary Polygons

Shamos shows that this is linearly reducible to line segment intersection detection. All the points in the two polygons are sorted in X. As in the previous algorithm these points are used as the base line for a scan line across the image. At each point the scan line has line segments crossing it and these are kept in an ordered list. At each step in the scan either a line joins the scan line list or a line leaves the scan line list. Furthermore if the line intersects with any other lines it will do so with *at least* its neighbour either above or below (in the ordering in which it joined the scan line list). As the scan line progresses whenever a line segment joins the scan line list an intersection check is made with its neighbours above and below. Similarly when a line leaves a list the new neighbours are checked. Note that in this manner the amount of line-line intersection checks is reduced, at the cost of an original sort which causes the overall run time to be $O(n \log n)$.

As Shamos points out, the algorithm will fail if 3 lines meet at one point, or if any line segment is vertical, and that

> " ...If either of these conditions is not met the algorithms we develop will be longer in detail, but not in asymptotic run-time."

This algorithm is quite acceptable as far at it goes but does not produce the intersection (let alone union) of the polygons, only an indication that it exists or not. Bentley et al in [BO79], taking this algorithm as a basis demonstrate an algorithm which reports (or counts) all the intersections of n line segments in $O(n \log n + k \log n)$ time, where k is, the number of intersections (note that this is not worse case optimal). This is done by the (in hindsight) simple expedient of processing the occurrences of intersections. When an intersection is found, instead of terminating the intersection is reported and the intersection point is added to the position of 'stopping' points in the event list (previously solely the endpoints of the lines). When such a point is encountered the two lines are swapped in the scan line list, further intersection tests carried out, and the algorithm continues. Berreta and Nievergelt termed this style of algorithm 'plane sweep'. Further details of scan-line algorithms are given in [NP82]. [LP84] gives details of scan-line and of related algorithms.

Scan line algorithms present a theoretically sound basis for a parallel merge system. No hardware architecture suggests itself immediately as suitable for parallel processing by scan line but it was hoped that some techniques could be developed which would be appropriate to other algorithms.

3.2 Parallelising the Merge Stage

As noted earlier the techniques used to parallelise any stage consist of selecting a theoretically sound algorithm and dividing up the function such that the limits upon parallel processing for P.G. are met - accuracy, speed, load balancing and memory requirement control.

3.2.1 The chosen algorithm

The chosen algorithm was Kilgour's Polygon Package. This was developed by Kilgour while working at Lattice Logic ltd. The package, which comprises of

about five thousand lines of IMP [Rob86], was productised at Lattice Logic by Neil Menzies. It is the at the core of the Lattice Logic Polymerge program and is used extensively in other parts of the Lattice Logic ShapeSmith products. The version which was taken as the basis for this development was 3.1.5.

This is based upon a scan line algorithm, but specifically aimed at integrated circuit pattern generation in that it carefully deals with cases which arise in VLSI pattern generation. For instance the case of abutting shapes is very carefully dealt with, as is the case of malformed (in the geometric sense) input.

The algorithmic performance of the basic algorithm is stated here without proof as being $O((n+k)\log p)$ where p, n and k are, respectively, the number of peaks (local minima) in the input, the number of input vertices and the number of intersection points. This is made up as the time to manipulate the event list (upto $O(k+n)$ operations on a queue of max size $O(p)$) and the edge list (similarily $O(p)$ operations on an edge list of max size $O(p)$) Note that the algorithm is not optimal in the worst (egg crate) case, being $O(n^2 \log n)$ (since p and k are $O(n)$), but it is optimal in the best (linear) case at $O(n)$ (since $p = 2$ and $k = 2$). The absolute performance is difficult to quantify since it is thoroughly implementation dependent (and even machine configuration dependent as noted later) and is constantly being improved. Furthermore the processing time is so large that finding enough resource to run benchmarks is hard.

The system has six stages which include the actual functioning of the scan line algorithm. Each stage needs to be analysed for the possibility of parallelising. The six stages are:

1. Parse the input format and flatten to the polygonal format.
2. Normalise the polygons.
3. Generate original event list.
4. Perform the scan-line processing.
5. Apply any sizing.

6. Decomposition (if needed) and output.

These are described to the extent that they affected the work detailed here in section A.1. Complete details are given in [Kil82] and [Kil86]. It is important to note that the output has no hierarchy associated - it consists of one cell containing all the geometry. In VLSI design 'hierarchy' has many definitions. To be able to deal with them the most general definition needs to be assumed. This in turn can cause computational difficulties [BOW83]. To avoid this all hierarchy is removed at the earliest stage. In this manner all the pattern generation algorithms are somewhat of the form of brute force and ignorance. Partial retention of hierarchy during the merge processing, with the attendant advantages of increased speed, for both the merge and the fracture stages, is a matter for further research.

Before describing the manner in which this basic algorithm has been developed to run on parallel processors, it is necessary to sidetrack briefly and examine the general nature of integrated circuit geometry.

3.2.2 Integrated Circuit Geometry

Forming any detailed classification of integrated circuit geometry is virtually impossible since designs will reflect the method in which the design has been generated, the philosophy of the design and the designer's teacher, which tools were used in the design and so forth. Student designs will be completely different from hand-designed products of a silicon design centre which will again be different from those produced by cell assemblers or gate-array generators. The difference in technology (CMOS/nMOS/bipolar) will affect the design geometry. [Whi85] gives some examples of 'circular' geometry. However more general classifications have been achieved.

Working at CMU, Bentley et al [BHH80] studied Manhattan designs of various sizes and extracted the proportion of shapes which were 'components' (small

rectangles), 'wires' (long thin rectangles) and other rectangles. Without reproducing the results in full here the percentages were, approximately 65%, 30%, and 5% respectively. These figures were used to influence design decisions when constructing a design rule checker. This excellent work is slightly restricted for development of pattern generation algorithms in that it solely deals with Manhattan geometry. Furthermore the input was non-merged, that is the only input was rectangles and no attention was paid to overlap. This was the correct decision for the design on a DRC system, but since there was no data about the results of merged systems it was considered useful to do an overall analysis of the *output* of a merge stage, to see whether any trend in the output could be used to influence the partitioning of function.

By doing edge counts on merged PG data it was discovered that the majority of output shapes were rectangles see Table 3–6. Of the remainder of the shapes the vast majority had less than 20 edges. In all cases there was a rapid drop in number of polygons as the edge count increased. In a very few cases were there polygons with greater than a thousand edges. In most cases the large polygons are accounted for by the power and ground nets on the metal layer, although one sheet with 40,000 edges on the active area mask of a CMOS design formed the transistor net of a PLA.

Obviously even with this small sample there is a variation between layers eg the contact and via layers will be mostly, if not exclusively, made up of large numbers of very small shapes - these are usually squares, but it is not unknown for them to be flashes - circles which have to be approximated as n-gons where $n > 8$. Similarly, as noted above, the active area layer in CMOS designs and the metal layers in all designs can have high vertex count polygons. The trend, however, remains.

3.2.3 Parallel Merge - How

In developing concurrent systems it is obviously highly advantageous to move as much function into the parallel part of the system as possible. There will of course be parts of the system which must remain as solely sequential.

As detailed in Section A.1 the code of the polygon package is very well organised with modules dealing with input, output, polygon normalisation, edge handling, path handling, event handling and so on. With some experience with the code, an obvious form of parallelism is to divide the function over processors - each processor (or processor group) dealing with one fixed function as previously handled by a module.

After consideration this technique was rejected for the following reason. There is obviously an upper bound on the number of functions which can be processed separately. When a certain number of processors have been added they will cost more in synchronisation and communication costs than they add by sharing the load. A requirement of any special purpose hardware dealing with integrated circuits is that it should, as much as is allowed by Amdahls law, be upwardly expandable with the problem. This is not possible with the system described above.

Further reasons for the abandonment of this approach was the communication costs envisaged and the fact that the topology of such a system was hardly likely to lend itself to the fracture stages and even less to other general purpose computing. A different approach had therefore to be developed.

The chosen approach

It was noted earlier that the output of the merge stage consisted of a very large number of simple shapes. It may thus be inferred that in the majority of cases an input shape will have interaction only with other shapes in its immediate vicinity. Thus if a processor was assigned to dealing with this shape there is only a low possibility that it will have to communicate in *any* manner with a

processor assigned to a shape some distance removed. As already mentioned the use of one processor per shape whether input or output is unrealistic. However the idea can be extended and a processor assigned to a *region* containing many putative output shapes. Thus instead of distributing the function over the processors the data is distributed with the (almost) complete function at each processor. There has been a payoff between memory and communication and the former has been chosen.

The now eight stage system was accordingly treated thus.

1. Parse and flatten. This must obviously remain a sequential process. Recent development has been addressed at removing the parse overhead from the merge stage. This is achieved by parsing in a separate operation to a disk-based parse data-structure.

2. Normalisation. This would, at first sight, appear an obvious candidate for moving into the parallel part of the system. Unfortunately before parallel processing can start the data must be divided. The division stage requires that the data be well formed (normalised). It appeared that the normalisation stage was required to be performed in the sequential part of the system. It was believed (and early experiments bore this out) that the time required by the normalisation would be minimal and so it would not be problematic. Later sections re-examine this decision and note ways of moving both the flattening and the normalisation process into the parallel part.

3. Division. A detailed description of this is given in section A.2. At this point all that concerns us is whether it is performed in the sequential or parallel stage of the processing. In the former case all the input shapes are divided, clipped where necessary, and passed individually to the processors. In the latter case all the shapes go to all processors which individually decide how much of which shapes is relevant to them. In hardware terms the former would be achieved by multiple point to point commu-

nication, whereas the latter would be achieved by a broadcast system of some sort. In performance terms the latter is obviously the better choice, since although more actual computation takes place it is spread over all the processors, thus reducing overall elapsed time.

4. Preprocess, Merge, Size.

   These three are the central parts of the process and as such are processed in parallel.

5. Recombination of data. Each processor will produce output shapes which may be classified by whether they are totally within the processor's 'region' or run along the boundary (indicating that this shape is part of a larger shape covering more than one region). In the former case the data may be output directly but in the latter the shapes should be combined. There are two ways in which this recombination could be achieved, either processors can negotiate the recombination between themselves or this can be left to a separate recombination stage (which may of course be in a separate processor). The former case requires processor to processor communication which would increase bandwidth and impose some sort of topology.

6. Decomposition and Output. As mentioned elsewhere decomposition is peripheral to this study. It is noted in passing that all decomposition can take place in parallel so long as care is taken that the recombination does not cause output polygons to be larger than the threshold size. Output is obviously a purely sequential operation.

3.2.4 The system architecture

Having considered a collection of techniques which should allow concurrent merging of integrated circuits, the problem of mapping these techniques onto a system architecture upon which it can run with minimum overhead needs to

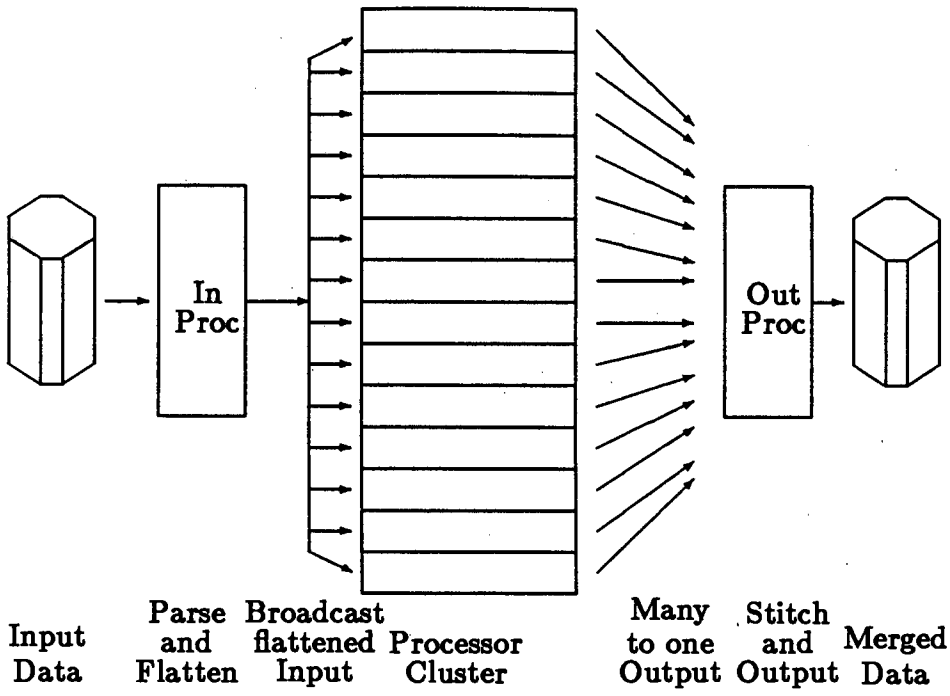


Figure 3–7: The Overall System Architecture

be addressed. The technique consists of a large parallel processing phase 'sandwiched' between two small, less immediately parallelisable, phases concerned mostly with input and output respectively. The communication from the input to the parallel stage is by broadcast, but that to the output is, at least notionally, on a point to point basis. Thus we have the broad outline shown in Figure 3–7.

The upward limit on the number of parallel processors is governed by the capabilities of the input processor, the output processor and the communication between them. Up to the limits the parallel processor can be expanded arbitrarily as design size grows. Ways of mitigating the restraining effects of the IO processors and the communication bandwidth are investigated later.

The mode of functioning is very much 'batch' oriented with a distinct input phase being followed by a separate merge and output phase. Thus the functions of the input and output could be put into one processor with only a

small loss in efficiency - this being due to the necessity of emulating broadcast communication on a point to point connection or *vice versa.*

3.2.5 Implementation

Although this approach seems intuitively good, it remained unproven and so a simple implementation was considered. There was no suitable multiprocessor system available and so the system was built in the form of a emulator. This approach had the advantage of flexibility; furthermore had the system been implemented upon real concurrent hardware the results would have reflected the hardware performance as much as the software. In the emulations the numbers of processors chosen were squares for the obvious reason of simplicity. A procedure 'Emulate Processor' was written which performed the function of one processor. Since each merge processor was performing the same operations the same procedure could be used for all. An integer parameter to the procedure indicated which area of the design was to be processed.

A special module was developed which handled the emulated time. This took as a basis the operating system provided clock. At the start of the emulation of each processor a variable was initialised with the current value of the system clock; at the end this value was subtracted from the clock, the value remaining was the time which that processor spent. When all the processors had been emulated the maximum time was added onto the notional current time. During any part of the emulation (either of a merge processor of during the sequential part) a current emulated time could be found by calling a function within the module, this feature being used extensively when measuring bandwidth. Given this basis, the emulation took the form of a simple iteration:

```
Perform input
Stop Sequential Time
%for Processor Count = 1, 1, Num processors %cycle
    Reset Emulated Time
    Emulate Processor (Processor Count)
%repeat
Restart Sequential Time
```

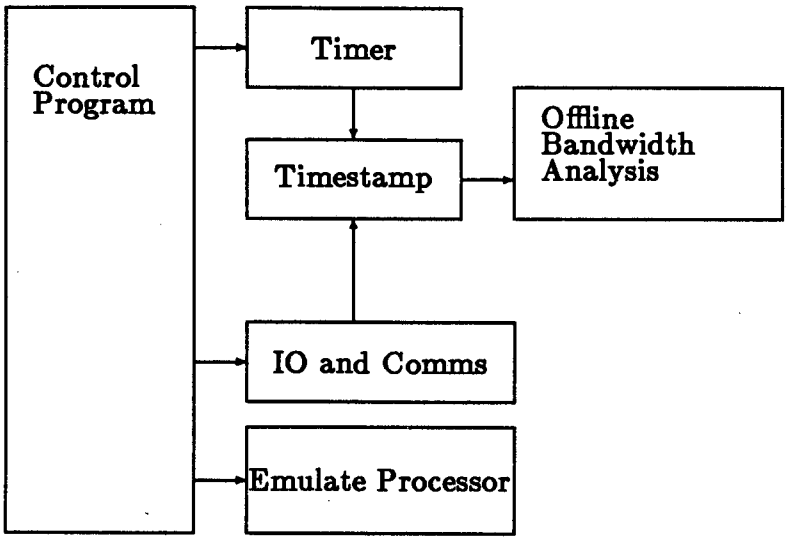


Figure 3–8: Pictorial representation of the harness

Figure 3–8 illustrates the conceptual construction of the harness. The control program, timing module and timestamp facility make up the harness, whilst the processor emulator and the IO and communication package form the system under evaluation. Obviously the latter form the major part of system in physical terms.

Such a system can of course only be an approximation to a real system, the more so since operating system provided clocks are notoriously inaccurate. It should be noted that all operations which are being emulated *are* measured by some clock and as such all timings will be representative. In fact the greatest problem in verifying such a system lies, not in ensuring that the times reported are accurate but in verifying that the output data is accurate. This topic is further covered in Section 3.3.1

As mentioned in Chapter 1 all timings given should be treated not as absolute, but relative to the base time - that taken by a single processor of equivalent power and in equivalent configuration. For this reason, when presenting the re-

sults, no special emphasis is placed on precise timing. Rather the emphasis has been on demonstrating trends.

During the first iteration of design no trace was kept of communication costs - communication was considered to be costless. This simplicity was justified in that at this stage the main effort was to produce a working system to which could be added software timing measurements in a similar manner to probes being attached to real hardware. At every possible point communication was minimised and it was hoped that communications costs would only marginally affect overall performance. As described in Section 3.3.5, once the system had been shown to work the communication cost was further analysed.

The new modules

A brief description of the modules used is given in Section A.2; SPLIT clipped the input shapes on a per-processor basis and STITCH recombined region-spanning shapes on output. In order for the emulation to function the main control module was substantially altered and a minor module (TIMER) written.

Coping With Sizing

When sizing is going to be applied care must be taken along the region boundaries that the sizing is done correctly. See Figure 3–9.

Shrinking In order that the data within one processor's region will be accurate once the shrinking has been applied, the area which the data is clipped to is increased by the amount of bias. Thus when the shapes are shrunk they are all precisely within the processor's region. Recombination can take place as for merge where there is no sizing.

Bloating Coping with bloating correctly is marginally more complicated and requires that convex angles be clipped.

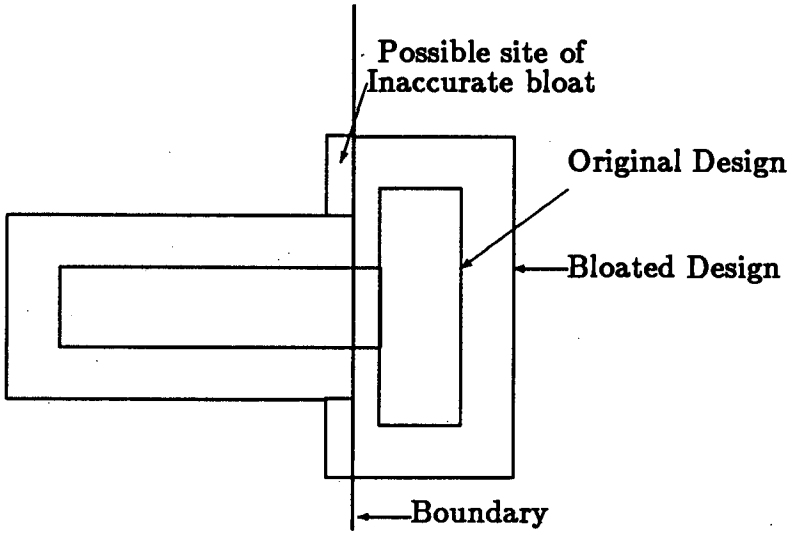


Figure 3–9: Incorrect sizing at Boundaries

Observe that the shapes within a region can only be affected on bloating by shapes within a 'bloating width' of the region's boundaries. The region is expanded as for shrinking, then merged, bloated and re-merged. So long as acute angles are clipped correctly all the shapes and parts of shapes within the *assigned* region will be represented correctly whereas the boundaries will not. The boundaries are clipped off and the shapes recombined as normal. The clipping is considerably simpler than that used in the SPLIT module, due to the known greater simplicity of the shapes involved.

3.3 Results, Modifications and Limitations

With this basic skeletal package in place the first round of experiments commenced. These consisted of emulations of processing clusters with the number of processors varying from 1 (the standard merge, used as the base line for measurement) to up to 49. These were applied, a layer at a time to a small-medium size P.G. benchmark, and several small university designs. At this stage in order to check the correct functioning of the emulated system the result was

checked against the original geometry by using the subtraction operator. Finally a limited number of similar experiments were carried out on one layer of a large design.

An important early result, which impacted more on further experiments than on the system design, was that, although the processing for a shrink operation takes longer than that for a bloat, which in turn is slower than a pure merge, any overall trends remain the same. Thus in further experimentation only a shrink or a bloat operation need take place; since they take longer any trend or timing disparity will be much more marked. See Table 3–7.

At this stage these experiments were not only taking timing measurements and so forth but also concerned with ironing out the 'wrinkles' in the system. Discounting convential 'bugs', these problems were in two main classes. Firstly there were problems caused by interfacing with the polygon package, such as making assumptions about the ordering of data. Very often the failure that such an assumption caused would not happen immediately, rather processing would continue until all the data was corrupt. The second class can be loosely described as rounding and rounding related problems. These would often lead to wrong ordering of vertices (for instance in the pre-stitch lists as described in appendix A). This in turn could cause program failure in such a manner as to cause difficulty with debugging. When dealing with a design with several tens of thousands of shapes what may be passed off as 'a million to one chance' suddenly becomes a distinct possibility! Furthermore it is in the nature of things that these problems only make themselves felt after several hours of processing.

3.3.1 Accuracy

The first notable result that these experiments showed was that the output may be different from that produced by the standard system, although never significantly enough to cause a design rule violation. The variations occur when non-Manhattan geometry is present at the boundaries and may be attributed to a combination of rounding problems and size representation inadequacies.

All positions in output formats are in integral measurements[1]. Thus there has to be an element of rounding.

Any difference is usually a collection of triangles along the region boundaries although a few pentagons and higher order polygons may be present.

Triangular Differences

This is both the most common case and the easiest one to allow for. It occurs when an angled line crosses a region boundary and its crossing point is calculated and rounded. Thus the re-joined polygon acquire an extra vertex. This problem may be easily overcome by filtering such extraneous points from the rejoined polygon, See Figure 3–10.

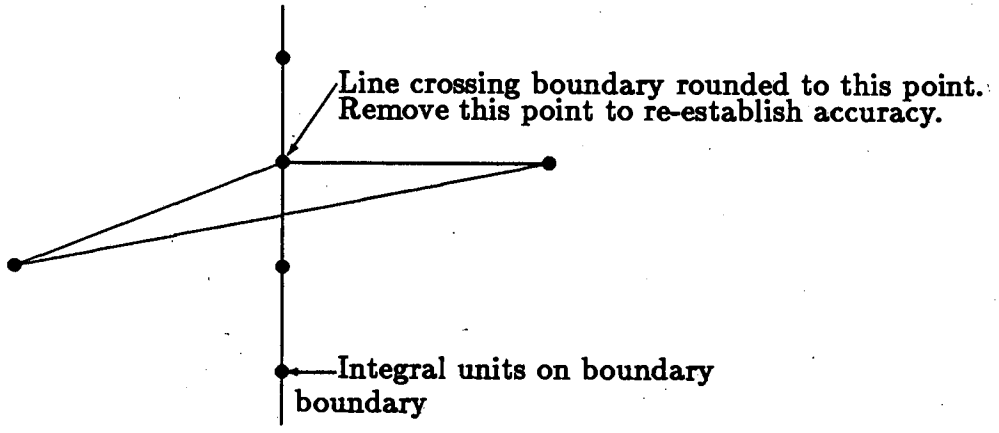


Figure 3–10: Triangular differences

Pentagonal and Higher Order Differences

Again this is caused by an angled line crossing a region boundary and an approximation of the intersection point being taken. If this line then crosses another the intersection point may be rounded down rather than up as previously, see Figure 3–11. Snapping edges will reduce the pentagon to a kite shape. The

---

[1]In the polygonal format used in this system this is one hundredth of a micron.

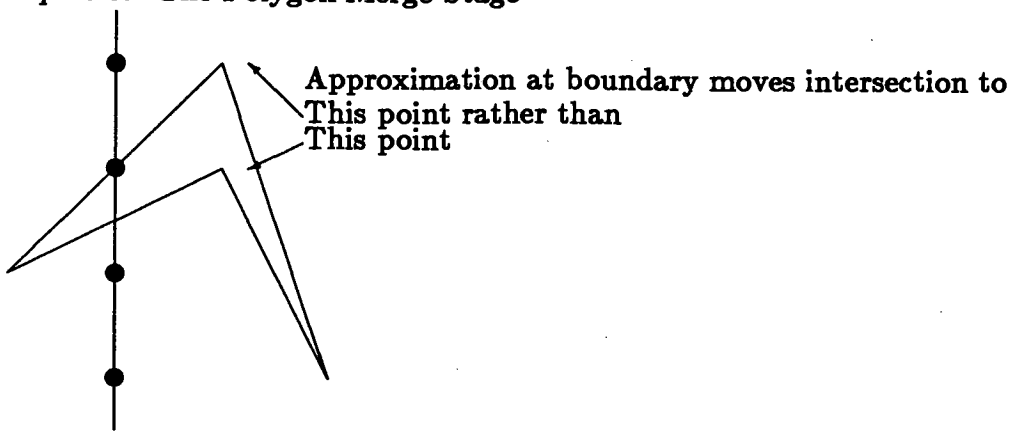


Figure 3–11: Pentagonal differences

vertex of the kite can only be inaccurate by a maximum of one representational unit which is considerably less than the tolerance of the mask making equipment.

3.3.2 Speed - Overall

The second result was at once the least expected, hardly relevant, and the most pleasing. In all the experiments the emulation of at least one processor configuration for every layer of every design took less time (real elapsed CPU time, not emulated) to run than the standard product. In many cases the difference was not significant, but nonetheless present.

It is interesting to examine why this surprising result happens. The time complexity of the merge operation is

$$O((n+k)\log p)$$

Which means for an arbitrary (but fixed) C.

$$Time \leq C(n+k)\log p$$

So assuming (simplistically) that the division is even into X areas which take exactly the same time to process the time taken to do the *merge* only is:

$$\begin{aligned} Time &\leq CX(\frac{n}{X}+\frac{k}{X})\log\frac{p}{X} \\ &\leq C(n+k)\log\frac{p}{X} \end{aligned}$$

This is an absolute improvement (of $(n+k)\log X$). If the speed up achieved thus is more than the time taken to run the SPLIT and STITCH modules then the overall time taken will drop and thus the absolute performance will increase.

Intuitively this speed up can be reasoned as the result of a divide and conquer operation taking place with a very cheap recombination stage. For instance in the calculation of the initial event queue, instead of sorting the minima of all the polygons in one set they would be sorted into several subsets, which will be cheaper. The topic of affecting the absolute performance of the sequential system is further discussed in Section 3.4.2.

3.3.3 Load Balancing

The third and the most important result, as far as the development of feasible pattern generation machines is concerned, lies in the time taken by each separate processor. The total time taken by the complete system is the sum of the time taken by the slowest processor at each stage and the (sequential) time take to perform the input and the recombination and output.

It is obvious from Table 3–8 that with 9 processors the overall elapsed[2] time is reduced by less than a factor of four. Although a certain amount of this inefficiency is due to 'data management' and is unavoidable the greater proportion is due to bad load balancing.

---

[2]split, merge and size and recombine

Reference to Table 3–9 shows that good load balancing is not being achieved - whereas some processors complete processing in less than a second one takes 60 second to complete (this is a very small university design used only as an example).

Achieving a good load balance

The bad load balance that the system shows is explained by the very simplistic method used for division. A regular grid is placed over the chip image and the regions are thus allocated. For this to work with any reasonable success the geometry within the chip image has to be distributed evenly so that the number of input vertices, peaks and output intersections in each region is roughly the same.

That integrated circuits do not have this property is not altogether surprising, even though it is commonly assumed that they do. Consider, for instance a region of the chip which contains routing. See for instance Figure 3–11. All the routing will take place on the metal and polysilicon layers, with a small amount of geometry on the contact layer. No geometry at all will be present on the diffusion layers upon which the routing will just appear as a gap.

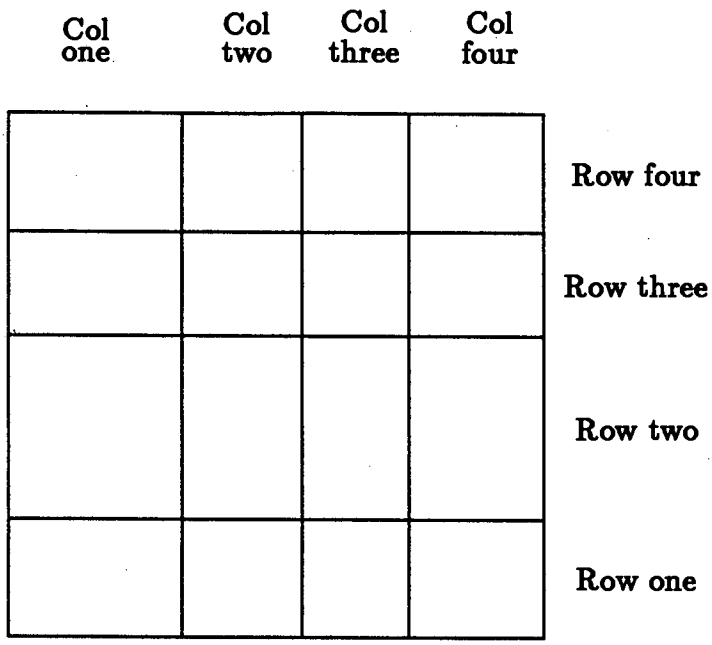
Examination of Tables 3–10 and 3–11, which show the vertex count within a thin row (respectively column) as the row (or column) is moved across the chip, shows the tendency for designs to become more dense towards the centre as noted in [BHH80]. This trend cannot be guaranteed. The best illustration of the non-uniformity is given by the pseudo-three-dimensional plot Table 3–12. The X and Y axes describe the chip image and the Z axis the vertex count (for one layer) within the small region around the appropriate X and Y. It is immediately apparent that the distribution is not even, with large areas with no vertices whatsoever. Even where there is a vertex presence the distribution is uneven.

With a 'perfect' division the time taken per processor to merge is constant. In theory the best way of achieving this is to equalise n (the number of input vertices), p (the number of input peaks) and k (the number of intersections). There is no simple way of gauging k or p from the input data, although it may be hoped that they will vary with n which can be measured very simply. Opportunity to perform this measurement arises both at the parse stage and the normalisation stage. The simplest measurement to take is the vertex distribution in X and Y (as shown in Tables 3–10 and 3–11), which may be done on the fly, during one pass of the data. Given this data it is a relatively simple process to make the divisions in X and Y such that each row *and* each column contain the same number of vertices. See Figure 3–12.

This will not give a completely even distribution of vertices in the enclosed rectangles (consider for instance a small dense area and its impact on the division). This is borne out by the timing results (Table 3–13), which, although better than for the simple case, do leave room for further improvement.

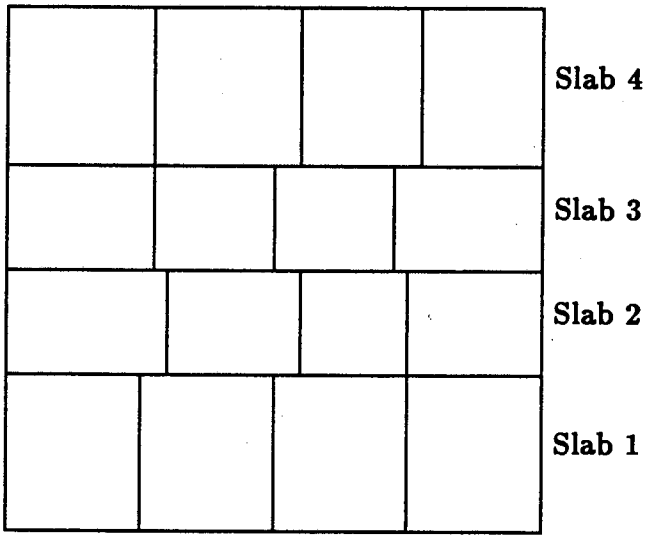
While investigating methods of achieving more even distributions it is important to realise that all the measurements will, perforce, be carried out sequentially. Any measurement should be cheap to calculate - every second taken in calculating the distribution is equivalent to one processor being one second slower than all the others. Thus overly complex region calculation can have a negative effect upon the overall run time.

The method adopted gives perfect division in terms of even vertex count which in turn gives reasonable results. This is achieved at the expense of an extra pass through the data and a marginally more complex recombination stage. As before the vertex frequency distribution is worked out in the Y direction and the design divided into horizontal slabs accordingly. On the second pass the X frequency distribution is calculated *within each slab*. Finally each slab is divided up into regions. See Figure 3–13. This gives much better results, reducing the idle time considerably (see Table 3–13).



Every row and every column has the same vertex count

Figure 3–12: Division into Y and X



Each region has the same vertex count

Figure 3–13: Division into Y and *then* X

3.3.4 Emulated Speed

Given that good load balancing has been achieved and that communication cost has been minimised the performance of such hardware should be optimal *for the chosen technique.* The effect of partitioning the design becomes more pronounced as the design gets larger. Thus the most efficient use of processors occurs with large designs. Tables 3–1 and 3–2 indicates this. Various layers of the base layers of a gate-array were processed on a variety of emulated systems. Due to limited memory on the emulating processor (and in fact any processor) the largest layers in some designs could not be processed. For each layer the vertex count V(an indication, as we have seen, of design complexity) is given and for each emulation the *emulated* time taken to process[3] the design is given in Table 3–1.

Layer	V	Norm.	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs	49 PUs
Active ‡	356452	3361	1361 *	1081	1034	1234	873	1040
Metal ‡	2089472	16069	5969 *	3875	3171	2365	2573	3086
Contact	2280949	17160	5178	2984	* 2380	1938	1897	1674
Poly	3726244	19161	6145	4006	* 3474	2928	3234	3106
Glass ‡	896	5	11	16	18	19	22	23

* indicates where the normalise and split stages become predominant

‡ indicates that the data contains an evaluation of the time to normalise
In either case the cost of the two pass area calculation is noted in the complete overhead.

Table 3–1: Emulated Time

[3]normalise where appropriate, split, merge, size and stitch

Table 3–2 gives for the same design the speedup for each processor configuration for each layer. Speed up is calculated as $S = \frac{T_1}{T_n}$ where T_n is the emulated time, n is the number of processors emulated and T_1 is the actual time for the uniprocessor configuration.

Layer	V	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs	49 PUs
Active‡	356452	2.5 *	3.1	3.3	2.7	3.8	3.2
Metal‡	2089472	2.7 *	4.1	5.1	6.8	6.2	5.2
Contact	2280949	3.3	5.8	7.2 *	8.5	9	10.2
Poly	3726244	3.1	4.8	5.5 *	6.5	6	6
Glass‡	896	0.45	0.31	0.27	0.26	0.22	0.21

* indicates where the normalise and split stages become predominant

‡ indicates that the data contains an evaluation of the time to normalise

In either case the cost of the two pass area calculation is noted in the complete overhead.

Table 3–2: Emulated Processor Speedup

In these tables the * indicates where the cost of normalising (if given) and split (which is a sequential process in this system) becomes predominant, that is they consume more than 50% of the emulated processing time. In sharp contrast with the output stage these stages cannot run in parallel with the merge and sizing. In this implementation of the system they must complete before the merging starts. The fact that for some layers the time given includes the normalise time and others not is a reflection of the fact that the normalisation and the division can take place separately (In some cases normalisation does not need to take place at all - if the input format is known to be well formed already such as for PG formats). In this case the 'base' system was treated equivalently (the time taken to perform the normalisation not incorporated). The high cost of these stages is due to the cost of working through the data multiple times, sequentially. In particular the normalisation and the area calculation are expensive.

The fact that these stages show themselves to be particularly dominant in these tables but not in Table 3–8 is due to the configuration of the systems from which these results were drawn. The processor upon which the measurements given in Tables 3–1 and 3–2 had around three times as much available real memory as the machine upon which the system was originally implemented. Lack of real memory has the greatest effects on the merge and size stages (consider the relative cost of page faults measured in millions rather than hundreds of thousands). Thus the speed up demonstrated in Table 3–2 is less than that which might be shown on a machine with less real memory. This in turn reflects upon the fact that any of these emulations are in fact emulations of groups of the *emulating processor*. Regardless of the reasons it is apparent that the overhead presented by the area calculation and division are candidates for removal by further development since they dramatically affect the speedup achievable. Furthermore although it *can* take place remotely it would be advantageous if the normalisation stage was encorporated as part of the complete system. Ways of achieving this are discussed in the conclusions and further in Appendix 2.

As can be seen from these figures, very often layers in IC designs are of such a small size that processing on large multiprocessing clusters is wasteful (and in some cases counterproductive). Having established which these layers are it should possible and might be advisable to divide a large processing system into two (or more) logical subsystems upon which the processing of two (or more) layers may be carried out concurrently. Indeed if the designs being processed are very small it should be possible for each processing unit to handle one layer of a design. Speedup by use of this sort of parallelism is discussed further in the final chapter. Assuming that the overhead of area normalisation and division are controlled, as designs become larger and larger the number of processors which can be effectively introduced will become limited only by the *communication costs.*

3.3.5 Communication costs

Communication costs are made up of two parts: the time taken by the output processor and the bandwidth between the processing units and the input and output processors.

Output Processor

For a given size of output the time taken performing the output operation will, to a first approximation be constant - *regardless* of the number of processors performing the merge. What is being measured is the time taken to write the merge data to disk together with any related processing. Since the output data is the same regardless of processor configuration, the time taken will be the same. There will be a slight extra cost involved in the handling of more communications. Table 3–3 gives, for each layer (in the same design as above), the time taken by the output processor and the number of merge processors in the configuration where the output processor takes as long as the slowest merge processor.

Layer	Output Time	Processor Threshhold
1	103	> 16
2	896	> 16
3	973	> 16
4	261	> 49
5	0.4	n/a

Table 3–3: Communication Overhead

This *processor threshold* which is obviously very data dependent will in turn govern the maximum number of merge processors. It should be added that during sizing and merging the output does not start until the first stage merge has taken place.

Bandwidth Cost

The bandwidth cost was measured by attaching 'software probes' to the system. Every communication 'event' was noted by writing an (emulated) time stamp to a log file. The size of the data transfer was noted with the time stamp. Off line analysis could then be made of the log. The graph in Table 3–15 shows how the communication cost varied over time for processing a layer from the example above. The communication cost is calculated as: $Cost = \frac{B_i}{T}$ where B_i is the bits transferred during the (fixed) time interval T. This is equivalent to the bandwidth required if all transfers were buffered over a period of T. The majority of the communication cost is for the output. Inspection of the graph shows that cost is not well balanced. However it is not high.

Tables 3–4 and 3–5 show, for 5 different processing configurations on each layer, the average and peak bandwidth costs respectively, where the average cost is the total number of bits transferred divided by the total (emulated time) and peak cost is the largest value of cost noted above where $T = 0.5s$. All values are given as kilobits per second. It can easily be seen from the tables that bandwidth does not represent an immediate problem. Furthermore even a 10 times speedup of the power of the merge processors would not cause problems.

Layer	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs
Active	17.5	18.7	19.5	17.3	20.0
Metal	12.2	16.4	18.3	19.1	25.2
Contact	20	26	28	30	30
Poly	20.4	25.6	27.0	28.4	27.3
Glass	1.7	1.4	1.3	†	†

Input

Layer	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs
Active	20.8	22.3	23.1	20.5	23.8
Metal	22.0	29.5	32.9	34.3	45.3
Contact	36	47	51	55	55
Poly	8.6	10.8	11.4	12.0	11.5
Glass	3.1	2.5	2.3	†	†

Output † indicates that no results were taken since they would have been meaningless.

Table 3–4: Average Bandwidth requirement

3.3.6 Processor Memory

This is an important aspect of the design by virtue of the impact of getting it wrong. If communication is underestimated then the system will merely run slowly - if memory is underestimated then it will not run at all.

Layer	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs
Active	191.2	190.1	191.2	188.7	190.4
Metal	59.3	57.6	57.6	57.0	88.0
Contact	90	88	88	87	88
Poly	513.6	513.9	511.0	520.0	495.3
Glass	5.1	3.2	2.9	†	†

Input

Layer	4 PUs	9 PUs	16 PUs	25 PUs	36 PUs
Active	260.2	541.8	596.8	1106	1204
Metal	107.7	200.0	349.1	543.2	1018
Contact	207	371	591	799	1115
Poly	558.9	789.5	1057	763.4	1236
Glass	73.7	105.4	62.2	†	†

Output † indicates that no results were taken since they would have been meaningless.

Table 3–5: Peak Output Bandwidth requirement

Using virtual memory for each of the processors in the cluster would, of course, solve this problem but there would be considerable extra complexity (and cost) in the extra hardware, local disks and so forth. There would also be a speed overhead in supporting the necessary operating system enhancement. Furthermore, assuming a complex system such as one with virtual memory control is not in keeping with the requirement of a non restrictive software architecture which will map onto many hardware configurations.

A series of merges, using the standard merge program was run and the memory requirement was plotted against the input vertex count. This graph is shown in Table 3–14 and shows a remarkably good fit of memory requirement to input vertex count. It is thus possible to judge the vertex count which the memory of each processor can handle.

3.3.7 Processor numbers

The limit on the number of processors is governed by many factors. For instance as can be readily seen in Table 3–1 when the time taken in area calculation and normalisation dominates the effect of adding extra processors beyond 9 is very small - but still achievable.

The fundamental limit upon this system is the size of the merge cluster at which merged data is being generated faster than the output processor can handle it - the communication limit. As can be seen from Table 3–3 this is very data dependent, but *for the given system* 20 would seem to be appropriate.

It should be noted that if the more time-consuming operations, involving sizing or the calculation of any non-intrinsic Boolean function, are performed then the elapsed time increases and the number of processors achievable will increase.

3.4 Conclusions

There are several important points which spring from the work represented by this chapter. First of all, however, it should be emphasised that there are quite probably great improvements which can be made purely be rewriting. This is discussed further below.

The techniques presented were developed as a result of studying the output from the system, not as a result of studying the algorithm. It should therefore be quite feasible to apply them to other algorithms which perform the same or similar PG operations. The method of speeding up is *independent of the algorithm*

The problems of implementing systems to perform merging are not dissimilar to those of implementing systems to do design and electrical rule checking. Indeed many of the algorithms are fundamentally the same. It is pointless to make predictions but it may well be that these two problems would be amenable to acceleration in a similar manner to that described above for the merge stage of pattern generation; indeed the decomposition algorithm described in the next chapter uses a scan line algorithm and would be amenable to parallelising as described here, if that processing presented a bottleneck.

The performance of the system when processing very large designs is difficult to gauge and can only be done by extrapolation from existing results. This is because of the difficulties of processing examples which represent state of the art and beyond designs. The merge stage of pattern generation is exceptionally CPU intensive - if it were not there would be no need to investigate concurrent hardware. Although in terms of CPU time the emulation of many configurations performing the merge stage of large designs could be handled (in a matter of CPU months) there is a critical shortage of memory.

The design of the emulation was such that the memory of every processor was kept in the (virtual) memory of the emulating host. Unfortunately this was limited to 70 megabytes and this is considerably less than needed to emulate *any* processor configuration when dealing with massive designs. The designs demonstrated here do however show that for this system the problem when dealing with large designs will be the output bottleneck.

3.4.1 Possible Improvements

The current system would probably be best implemented as having 20 merge processors, at which stage something around a 6 time speedup should be achievable. Although this is quite acceptable for a first pass implementation it is obviously desirable to bring the latter figure closer to the former while increasing the former. These two aims may be met by, in the first case, reducing the overhead of the area calculation (and if possible incorporating the normalisation stage) and in the latter case improving the implementation of the output system. By doing this a speed up well in excess of ten times seems quite possible.

Design Flattening, Normalisation and Partitioning

In the current system this takes place in the input processor. As noted in section 3.3.4 the effect of the calculation of the partitions can be large when very big designs are processed and thus there are gains to be made from improving them. There are two methods of approaching this, firstly improving the code efficiency and secondly moving as much function as possible into the parallel stage.

Improving the code is a standard software engineering practice. That the original code is amenable to improvement is demonstrated by the fact that a 10% improvement in the area calculation stage was achieved with remarkable ease. It has recently been shown that there is no need for the sub-areas to have a similar aspect ratio to the input data. Indeed there are certain cases when long thin data areas (with equal vertex counts) are preferable. Thus the second pass through the data prior to division could be avoided. By using these two techniques the load placed by the partitioning could be sizeably reduced.

Moving functions into the parallel part of the system is potentially much more effective in terms of elapsed time. In fact perfect hindsight provides a far better system in which the functions of flattening, normalisation, area calculation and division are all carried out in the parallel part of the system.

Each processor receives the complete data structure which results from the parsing with any hierarchy being intact. Each processor calculates the vertex density within a small region of the whole design. A small amount of interprocessor liaison establishes the merge region boundaries for each processor which can then flatten and normalise within its region only. Since each processor only needs to deal with data within a window the flattening then become much faster (when flattening hierarchical designs, cells whose bounding box are outwith the processors window need not be flattened). The normalisation is then placed within the SPLIT module. As each processor receives the (malformed) input shapes it decides by simple bounding box calculation whether they are relevant. Relevant shapes are normalised and then clipped.

Although every processor has to do slightly more computation than strictly necessary, since region-spanning shapes will be normalised in every relevant processor, the overall time taken will be considerably reduced and, more importantly, there will be no sequential overhead to swamp the time taken by large multiprocessing clusters.

Improvement of output

Although some of the function of output processing (data compaction and so forth) can be migrated to the parallel part of a system, ultimately output cannot become parallel and, by virtue of the data explosion which happens when flattening occurs, presents the ultimate bottleneck where Amdahl's law can no longer be put off. As discussed earlier for the given system this bottleneck occurs at 20 merge processors. At this stage a 5% improvement in the efficiency of the output (or migrating some of the function to the parallel part) means that the merge cluster can be increased by one. Chapter four demonstrates 7% improvement in output processor efficiency achieved just by moving a few lines of code. The output stage must be made as efficient as possible.

Changes to the basic algorithm

Although there are several improvements to the basic scan line algorithm they are not relevant here and are thus not discussed. See for example [OW86].

Further hardware acceleration

This architecture performs the pattern generation computation very fast. It should be possible to drive the fastest pattern generation hardware. If further speed is required this would need to be achieved by making improvements at other levels of the system hierarchy. A traditional way of achieving further speedup is by migrating function into the hardware. One way of achieving this has been briefly discussed earlier. The most obvious candidate for hardware acceleration is a line segment intersection calculation.

All the above are really what might be described as second and later generation developments. Their precise impact has not been studied, indeed to implement some of them, it becomes necessary actually to build an example system upon which to carry out further experiments.

3.4.2 Sequential Program Enhancement

It has already been noted that the emulations took a similar time to the standard Polygon Package merge for small and medium designs. For large design the timing differences are rather more pronounced. One large design took 8 hours to merge and bloat using the emulator with an elapsed time of approximately 14 hours. The Polygon Package required 12.25 hours and 38 hours elapsed. Although some of this astonishing improvement is due to the algorithmic improvements mentioned earlier the most part is related to the memory requirements. The Polygon Package requires 28 Mb of (virtual) memory and the emulator 19 Mb. The scan line algorithm has very bad locality of reference, especially on the second merge used when sizing, and thus reduced memory requirements will reduce paging requirements. This is confirmed by the emulator having 4 million page-faults against the Polygon Package's 13.25 million.

It is a characteristic of VMS (the operating system upon which all the experiments were carried out) that at least part of the cost of handling a page-fault is credited to the user process and thus the disparity in times become even more pronounced. Recently this memory control facility has been usefully exploited. Designs which could not be processed in 70 Megabytes can be processed in under 6 [Fer87].

In a survey of LSI artwork-analysis and design rule checking programs contained in [Bai77] Baird notes that there was a "widespread" use of partitioning (eg [Cra75]). This is a very similar operation to the splitting used in the emulations. Although motivated by a desire to conserve the use of store (most, if not all, of the described systems ran on 16 bit machines) it was noted that run-time could be "dramatically improved by the introduction of partitioning". The algorithms used had a time complexity of n^2 so the performance improvement was rather more marked. Baird shows, in fact, how with careful choice of partition size the growth can be made linear with the size of chip, although this cannot be true in the worst case.

The introduction of splitting to the polygon package was motivated neither by a requirement to reduce storage, nor run-time although both were achieved. In distinction to the partitioning Baird described all the data is kept in store.

Theorist are examining methods of reducing storage requirements [OW86]. Partitioning is a simple method of achieving this. Although most current generation VLSI pattern generation (and possibly artwork analysis) packages no longer use partitioning, as packing density and consequent complexity of designs increase partitioning will become more and more important. This will become true for all tasks which make up VLSI design.

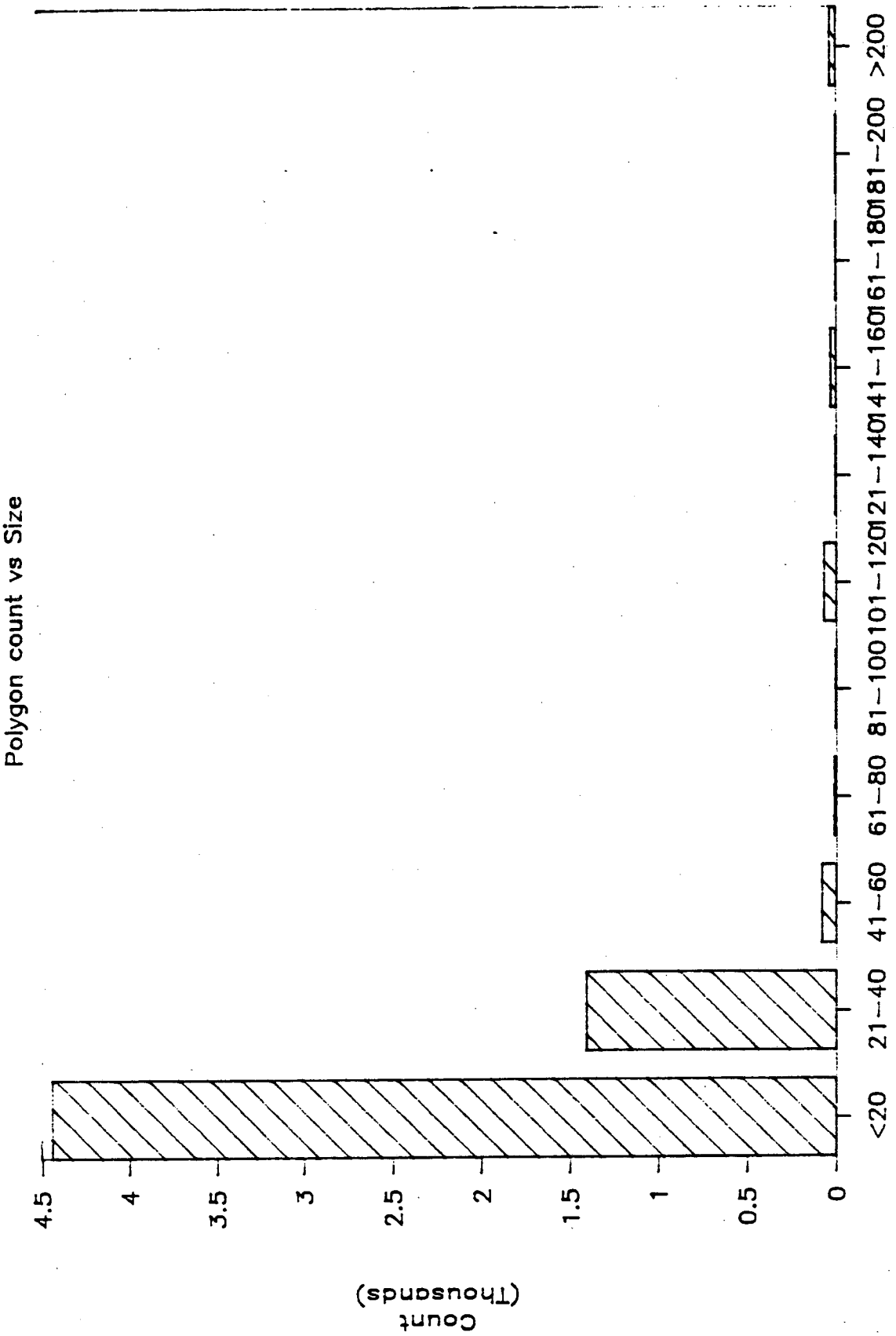


Table 3–6: Distribution of vertex count in merge data

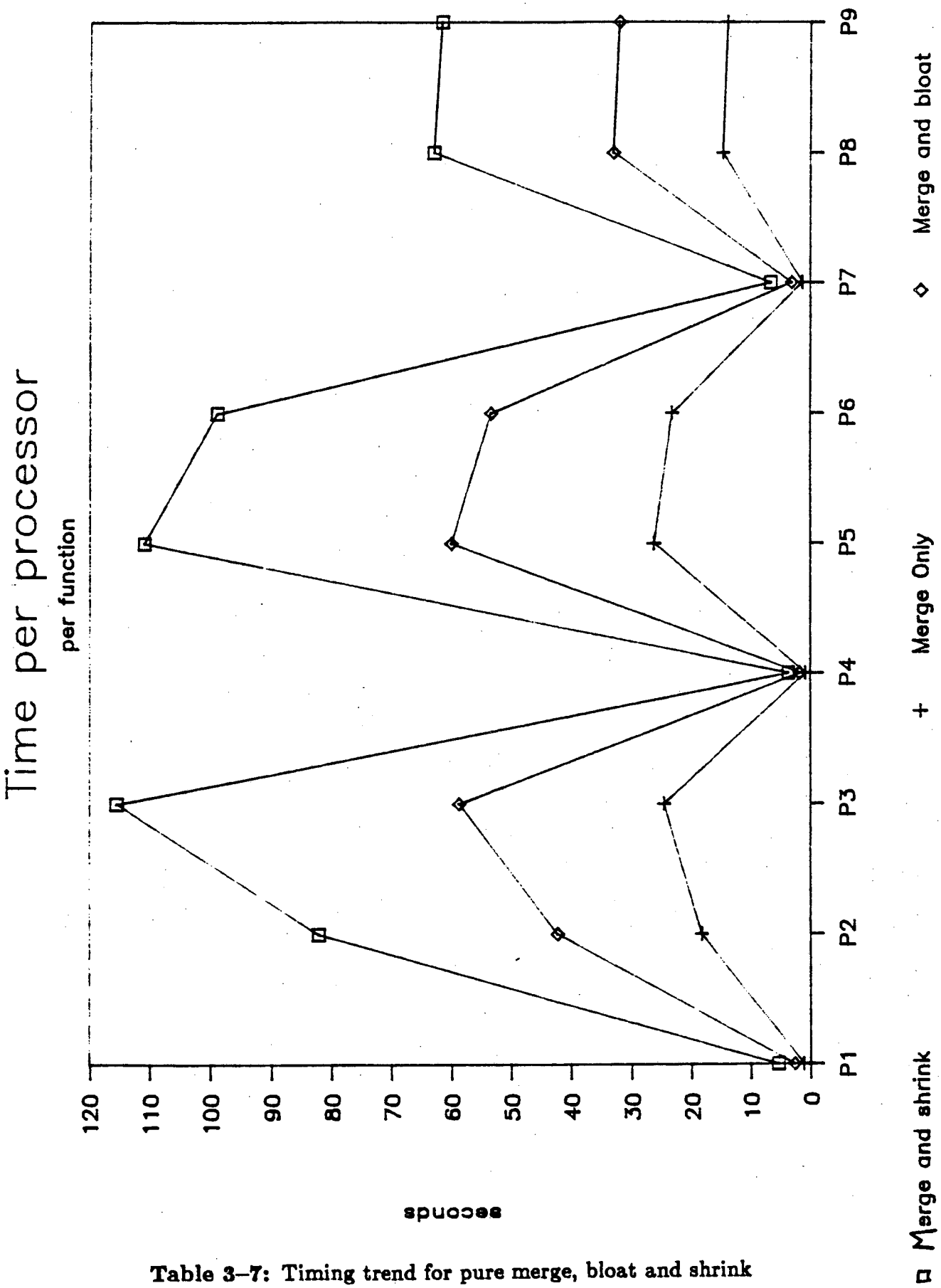


Table 3–7: Timing trend for pure merge, bloat and shrink

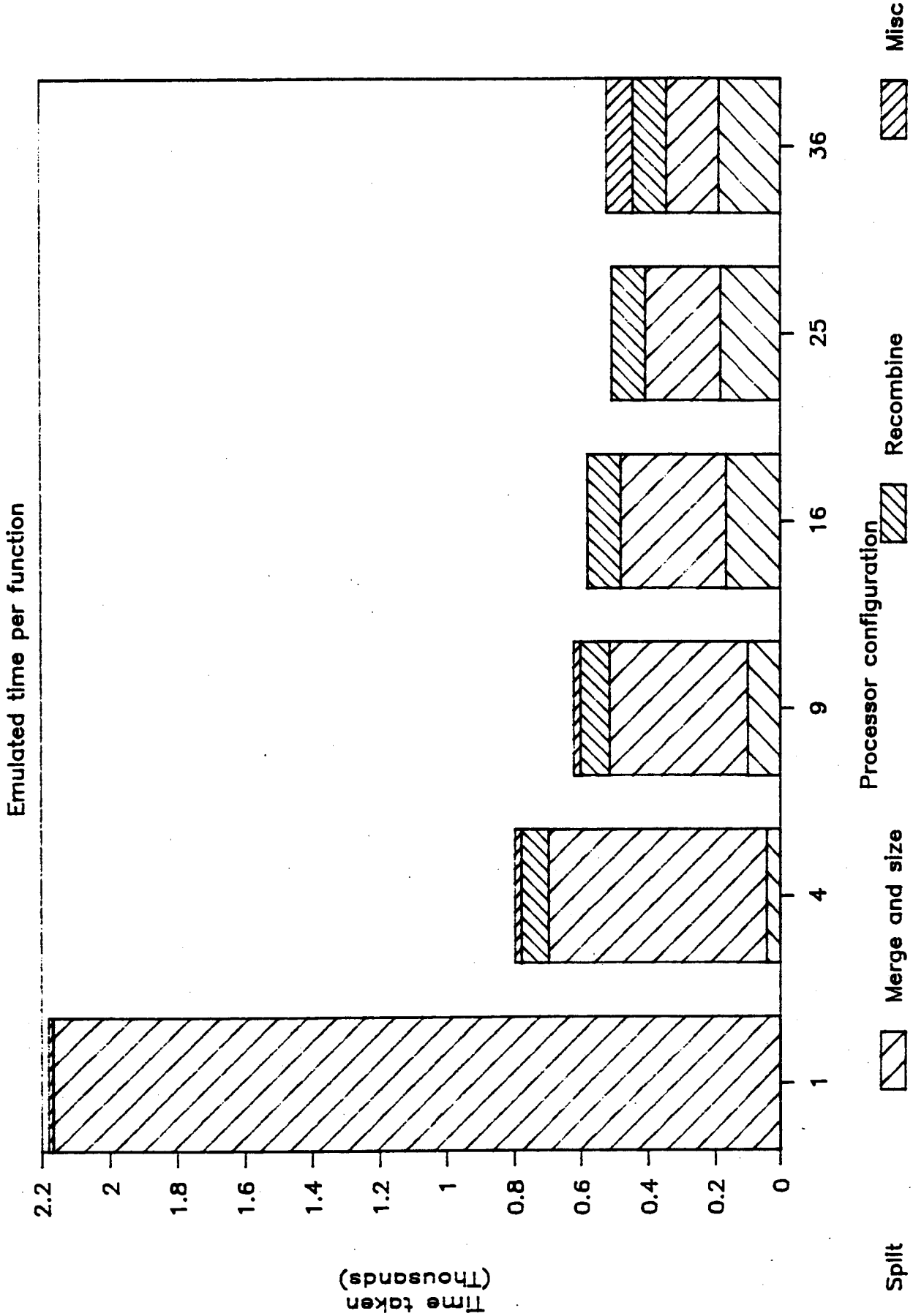


Table 3–8: Function versus Time

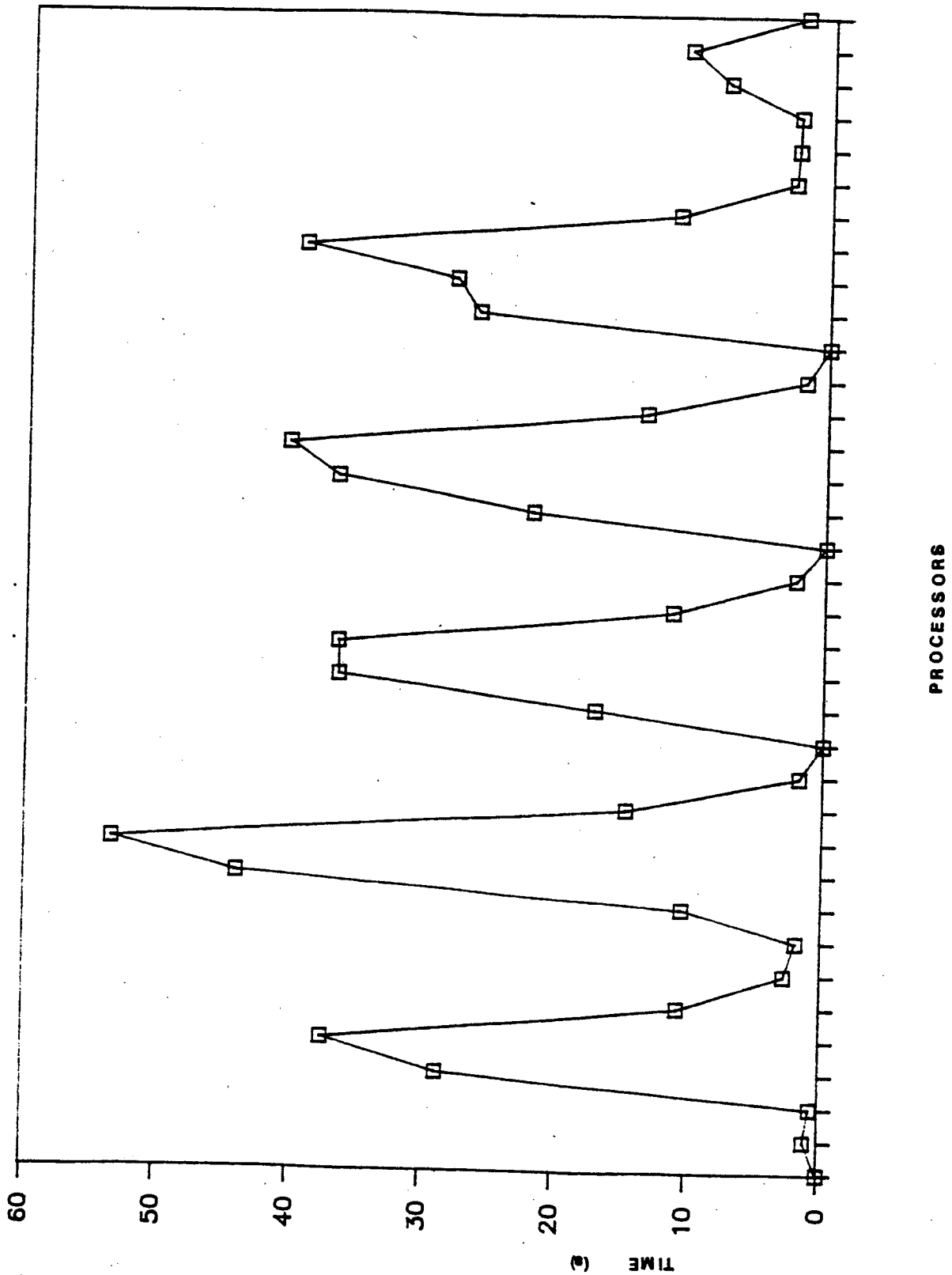


Table 3–9: Time Taken per Processor-Merge stage only

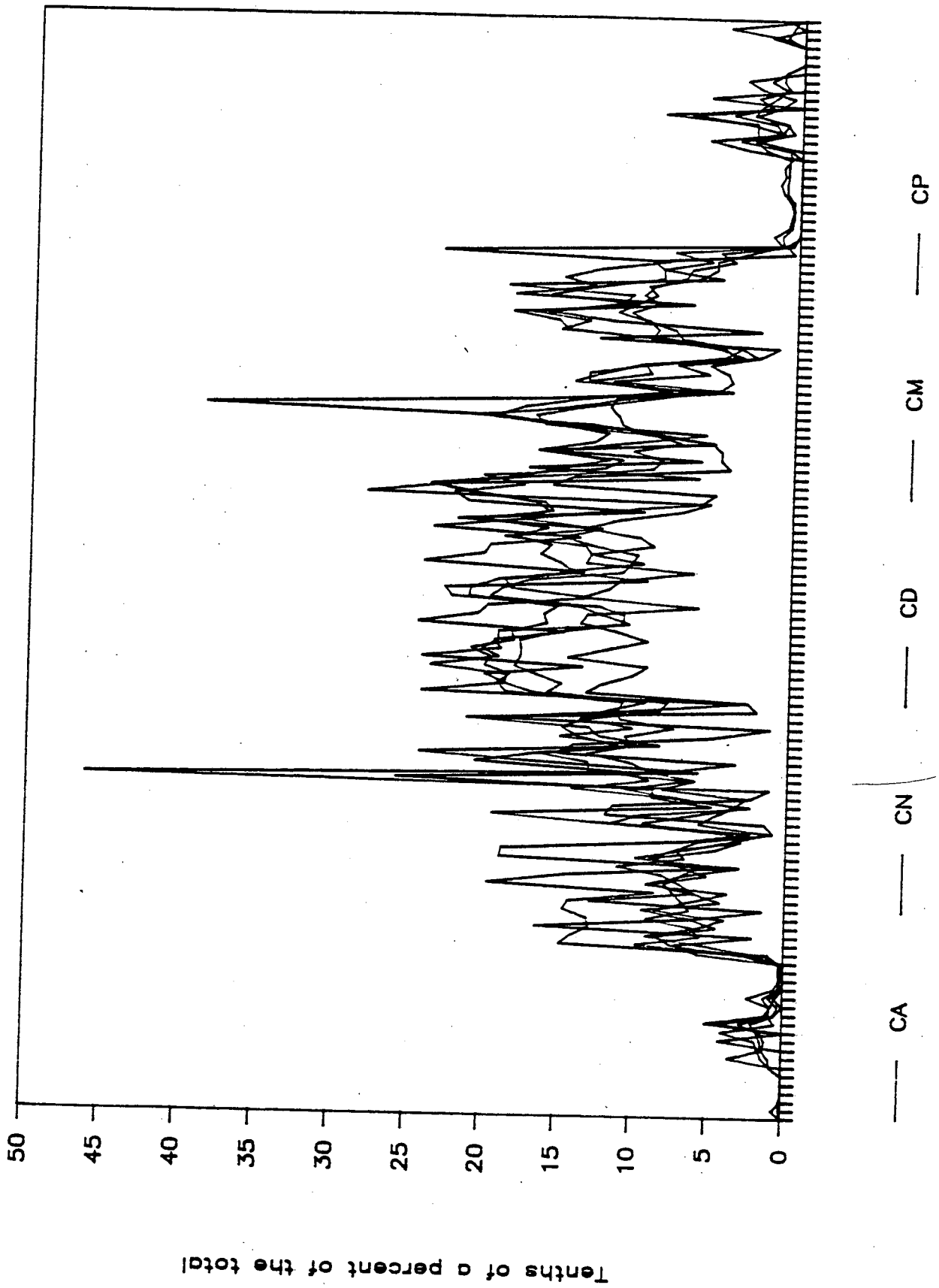


Table 3–10: Vertex Distribution in the X direction

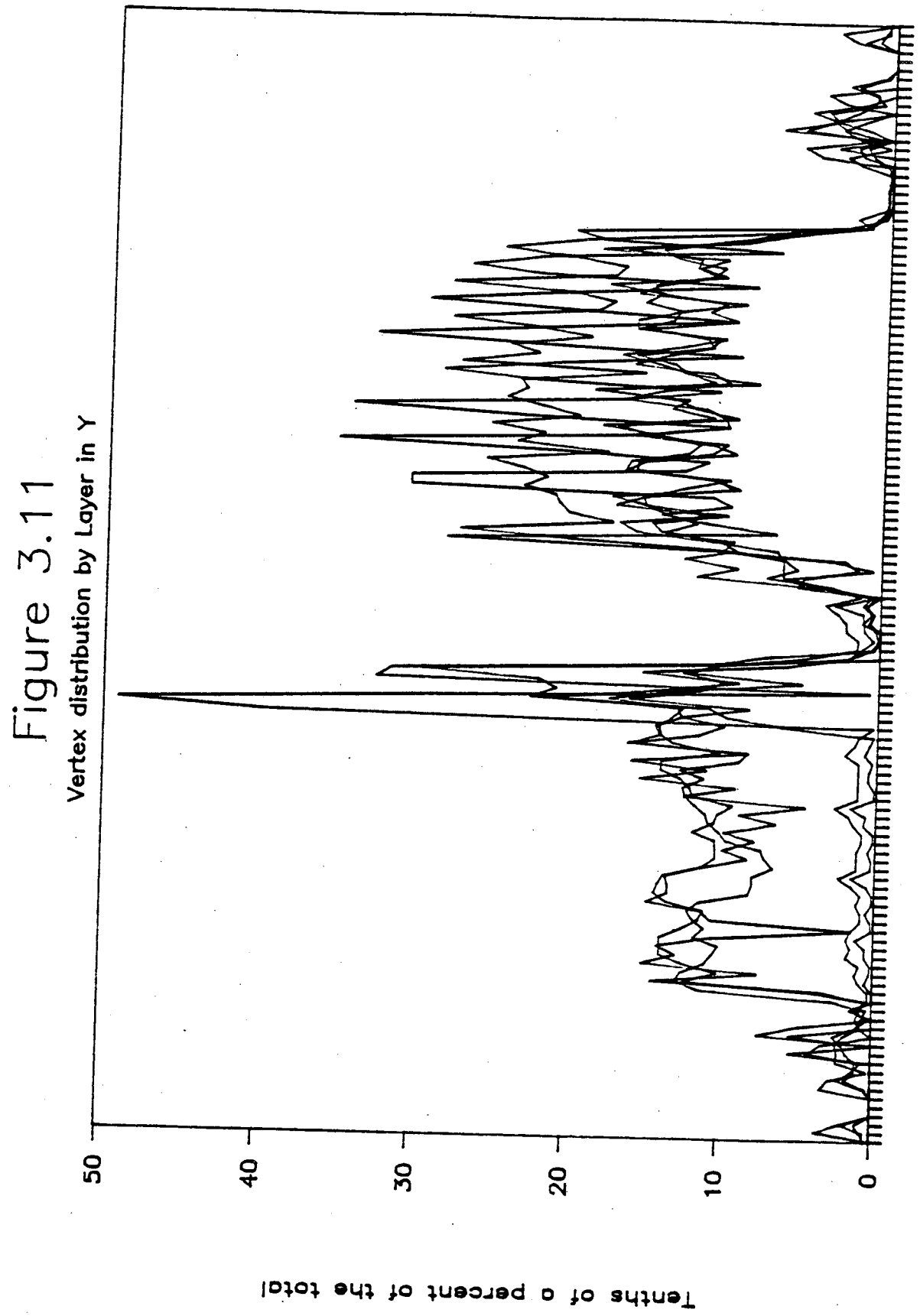


Table 3–11: Vertex Distribution in the Y direction

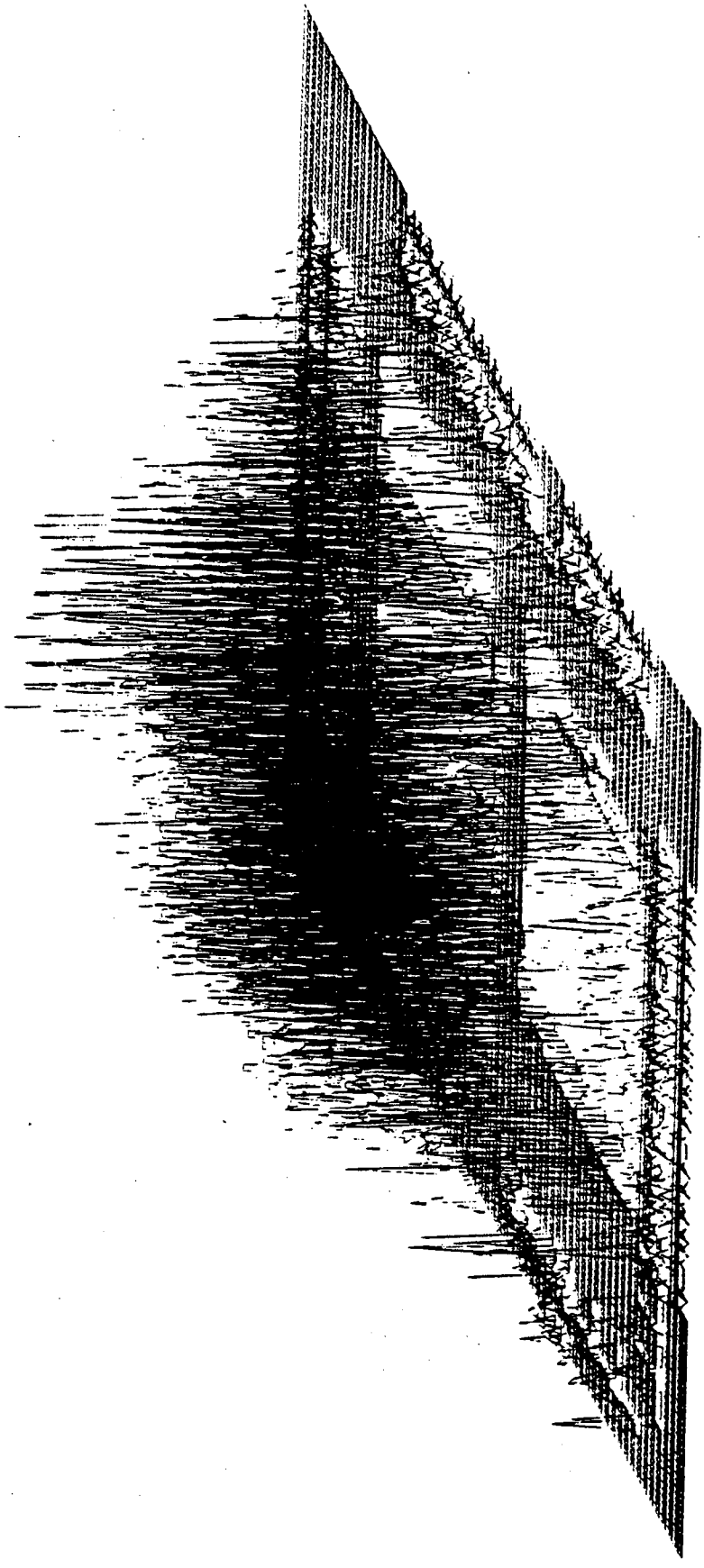


Table 3–12: Vertex distribution over area

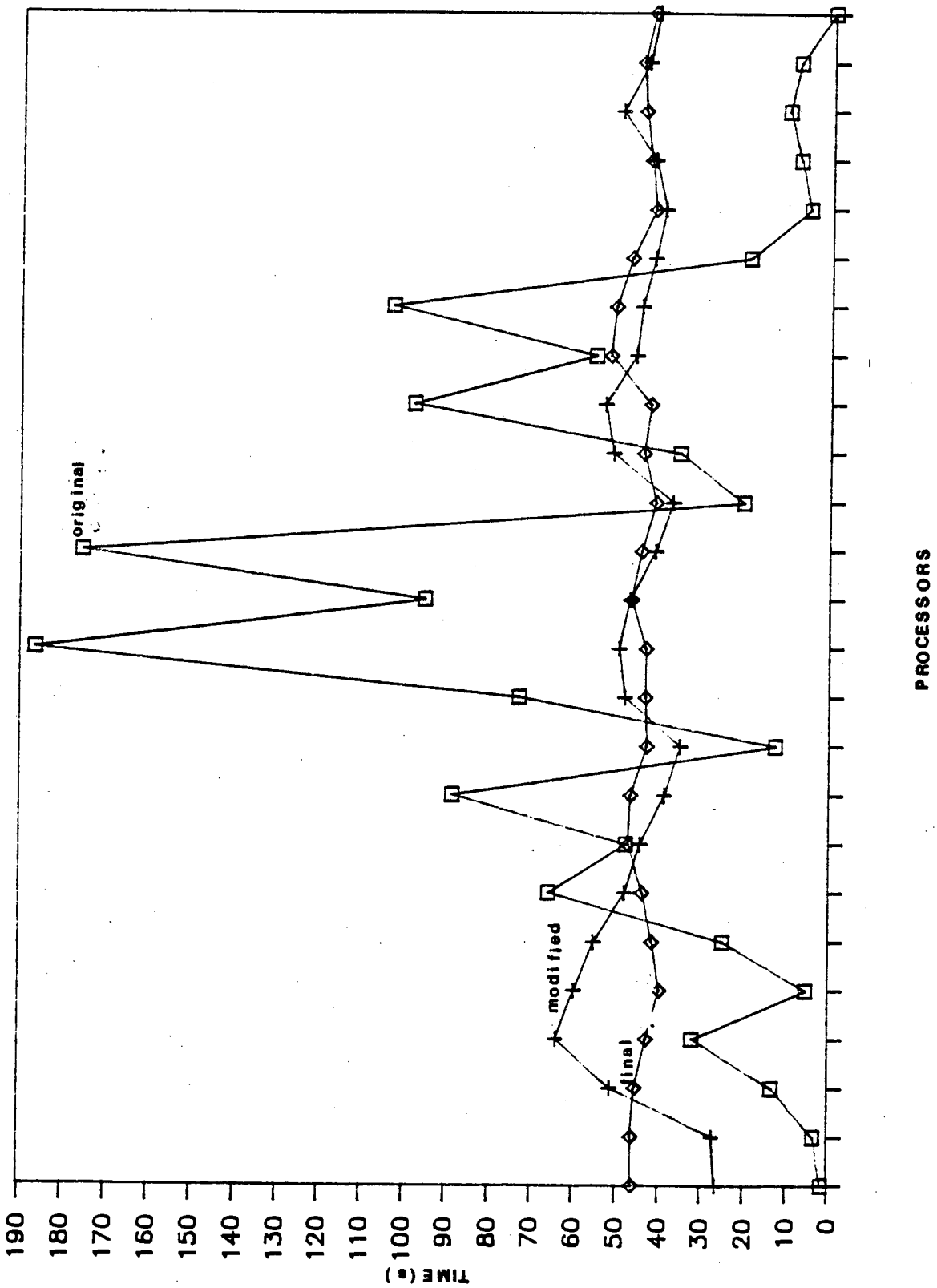


Table 3–13: Timing by the three division methods

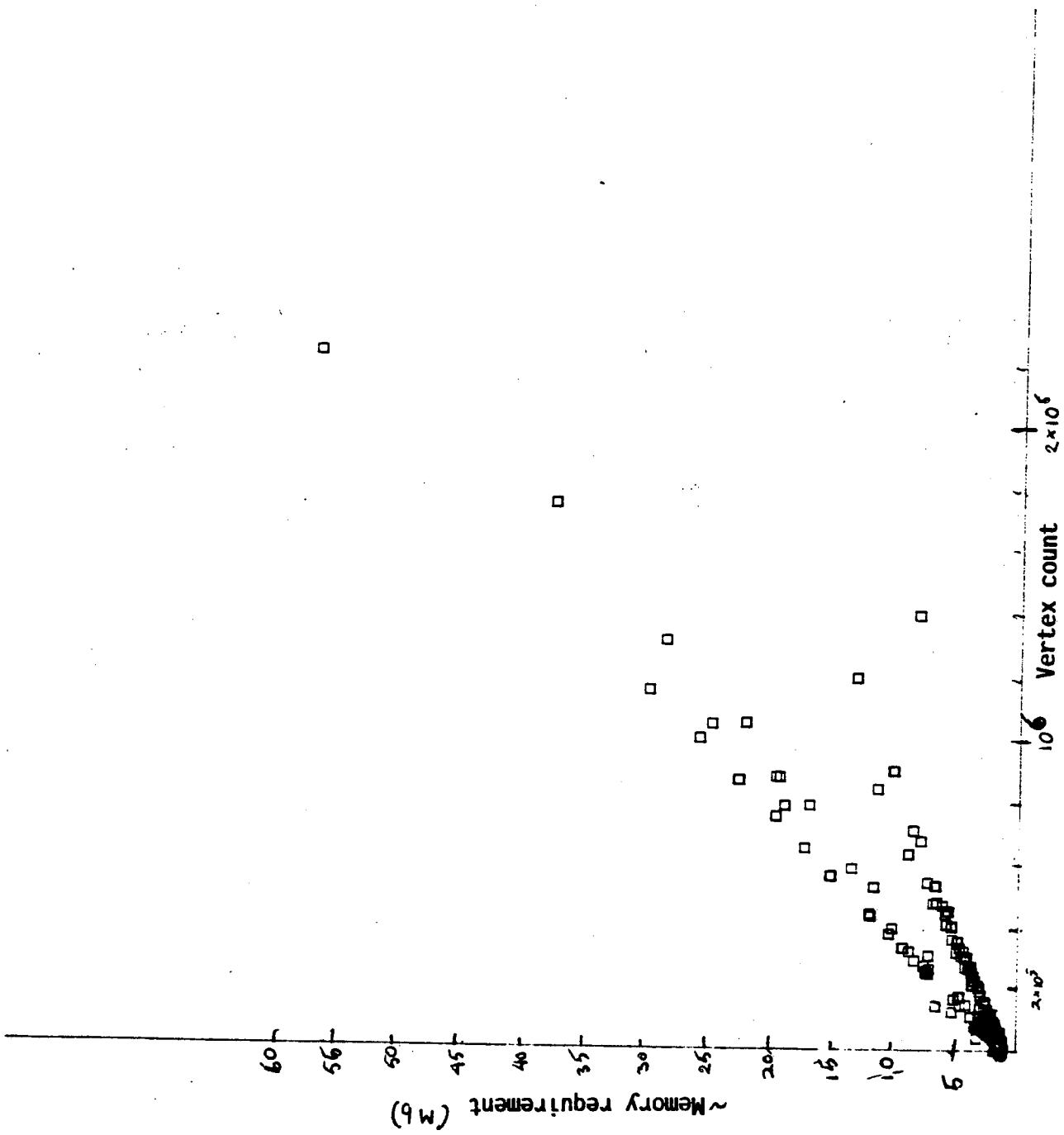


Table 3–14: Memory Requirement vs vertex count

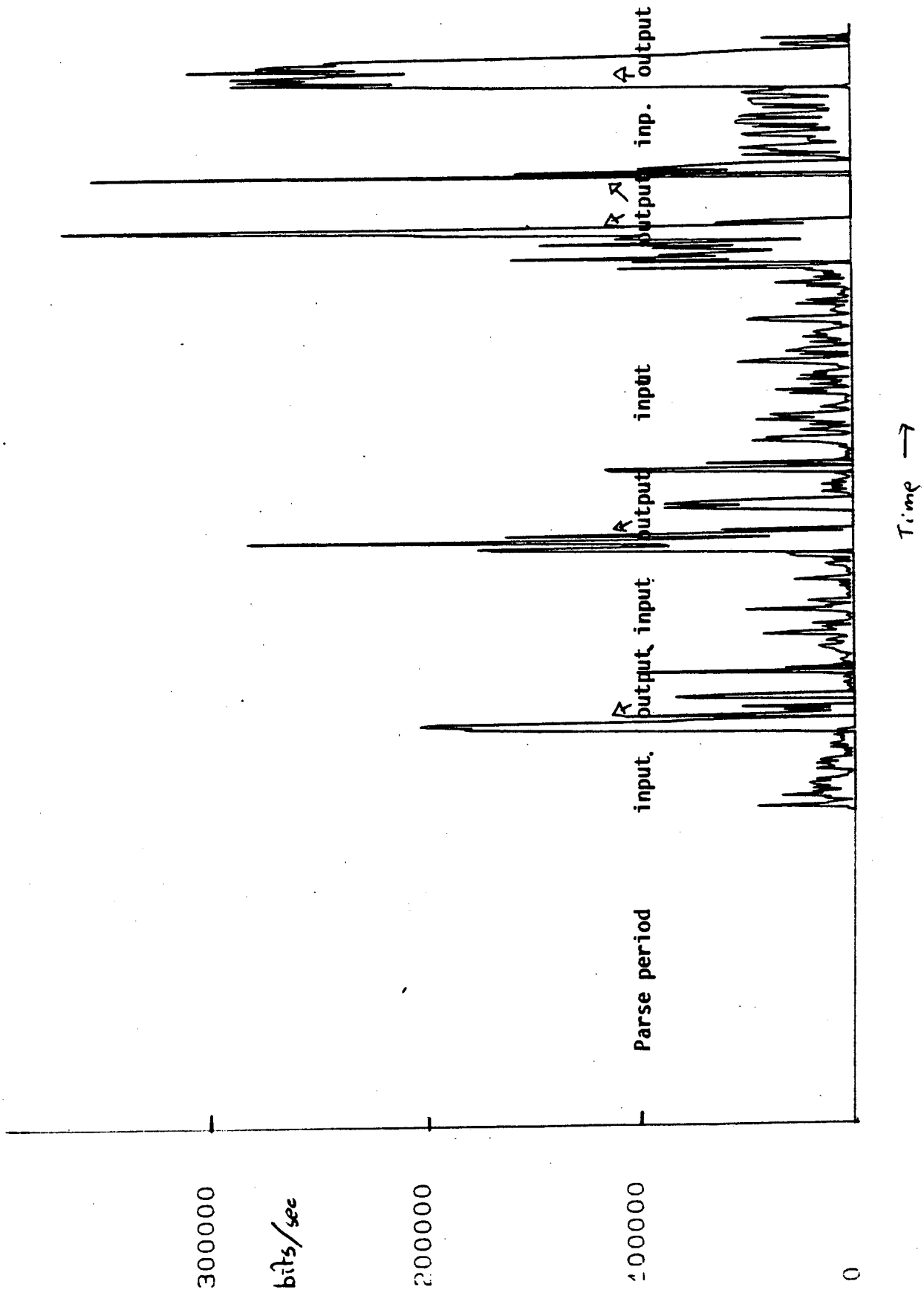


Table 3–15: Communication Costs for merging multiple Layers

Chapter 4

The Electron Beam Fracture Stage

The time taken to fracture polygons for writing by Ebeam lithography is very little compared with that taken to generate the merged and flattened polygons. It is however useful to spend some time examining ways in which the process can be parallelised.

4.1 Algorithms for Ebeam Fracture

As mentioned in Chapter 2 the input for electron beam mask makers (as well as the newer generations of other pattern generation equipment) is a series of stripe data subfiles, each of which consists of a list of trapezia whose parallel edges are horizontal. Performing the conversion *fracture* from the polygonal format output by the merge stage to this format is therefore a two part process: the decomposition of the polygons into trapezia, and the sorting of the trapezia into stripe data subfiles. It should be emphasised that the order in which these operations are performed is unimportant, what matters is the result.

4.1.1 Theoretical Consideration of Decomposition

Decomposition of any polygonal structure consists of finding a set of chords which break it down into a set of simpler structures. In [AAI86] Asano et

al investigate the computational complexity of the minimal decomposition of polygonal structures into trapezia - that is the decomposition which yields the least number of output trapezia. The approach they chose is to search for the maximum number of *effective chords*. These chords are used to divide the input polygon up into sub-polygons which may be easily further decomposed into the *minimal set of trapezia* by the scan line method given below. By reduction through various set and graph theoretic arguments they demonstrate that the searching for the maximum number of pairwise independent effective chords is *NP* complete and thus that the minimal decomposition problem is *NP* complete.

Working from the same basis - that of finding the maximum number of pairwise independent effective chords they then demonstrate an $O(n^2)$ algorithm for polygons with n vertices but without holes and extend this to an $O(n^{2+h})$ algorithm for polygons with n vertices and h holes.

As has been seen earlier the size of polygons can be quite vast and so $O(n^2)$ and higher orders of growth on the complexity of the fracture stage should be avoided if possible. Thus they present an approximate solution to the problem with performance of $O(n \log n)$. This approximates the finding of the maximum pairwise independent effective chords.

4.1.2 Scan Line Algorithms

The algorithms mentioned above are somewhat complicated, which is liable to affect their absolute performance (and, to an extent, reliability). Furthermore they all rely on an algorithm which will decompose the sub polygons into their component trapezia. This section describes such an algorithm which is a development of a scan line algorithm.

As described in the previous chapter scan line algorithms [NP82] work by sweeping a line across the object, stopping and performing various functions at key points (events). At any point the scan line is a list of those edges which cross the line at that point. The scan line, in conjunction with the list of pending events, controls the further operation of the algorithm.

For Ebeam fracture the scan line algorithm is modified as follows. The input is the polygon which is to be decomposed, the events are the vertices and the edge list is the list of those edges which currently cross the scan line. The number of edges is bound to be even and since the input is well formed they form *edge pairs* which always enclose the field of the polygon.

Processing an event has two major functions. Firstly, as for all scan line algorithms it causes the edge list to be updated and secondly it controls which trapezia are output. Whenever an edge ends (ie a vertex is reached) or a vertical concavity causes two new edges to be introduced between an edge pair, a trapezium is output whose upper edge is the current Y value of the scan line, whose lower edge is the last Y value for the edge pair and whose non-parallel edges are derived from the edge pair. Thus all the chords introduced by the scan line algorithm are horizontal and are introduced at events. See Figure 4–1.

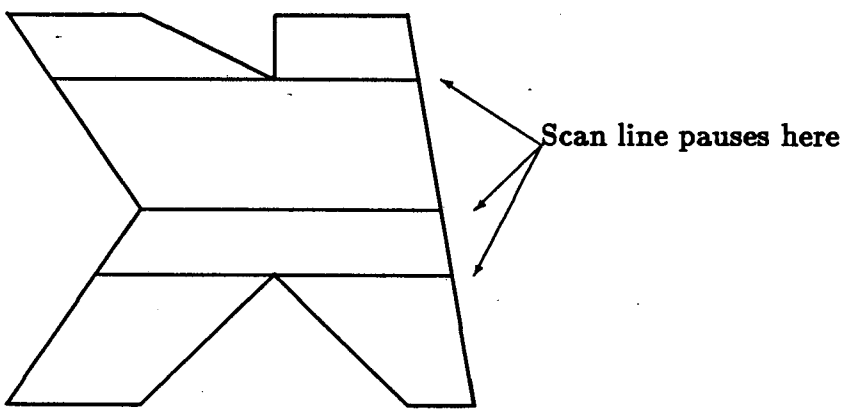


Figure 4–1: Outputting trapezia from a Scan Line Algorithm

The initial event list is the sorted list of local minima in the input polygon. Thus the time complexity of decomposition of a polygon with n vertices of which p are minima is obviously $O(p \log p + n)$. Thus the best case complexity

(for convex polygons) is $O(n)$ and the worst case (chicken feet and egg crates) is $O(n \log n)$.

4.1.3 Sorting

Not only do the input polygons need to be decomposed into trapezia, the trapezia have to be in a sorted form, such that all the data for one stripe is given with one subfile and that the subfiles are specifed in the correct order for the Ebeam machine.

The sorting can take place either before or after the polygon decomposition stage. In the former case the input polygons are sorted such that all the polygons which will contribute trapezia to that stripe, and thus subfile, are kept together. Since some polygons will span stripe boundaries there will be an element of duplication. Having been thus sorted, the polygons are fractured a stripe at a time. This generates the subfiles in the order required by the Ebeam machine. Any trapezia which cross stripe boundaries are clipped to the current stripe boundary.

4.2 Parallelising Ebeam Fracture

The development of Ebeam fracture to run on a parallel architecture is considerably simpler than for merging since there is an obvious operation which can be made to run in parallel - the actual performance of the decomposition.

4.2.1 Parallel Ebeam Fracture

Sequential processing requires that all the data is in memory before processing starts. This governs the order of processing of the individual polygons and takes place totally in isolation from the processing of previous or later polygons.

In theory, if the requirements upon the ordering of the processing of the polygons were relaxed, all the polygons could be fractured simultaneously, assuming that sufficient hardware was available; the total time taken being the time to fracture the most complex polygon. Thus very high levels of concurrency would be feasible. This restriction is only one embodiment of the fact that the pattern tape must consist of consecutive stripe data fields, each of which contains all the data for that stripe. If another way of guaranteeing this fact is adopted the restriction can be waived.

On first inspection it appears that the only way of assembling stripe data in the correct order after a random order fracture is to *sort* all the output trapezia. Since there may be millions of output shapes this would be prohibitively expensive on any hardware configuration. Examining the problem rather more closely shows that the stripe data need not be ordered, and so any sorting need only establish into which stripe file to place the trapezia. This can be achieved by careful file manipulation in a manner similar to the first pass of a radix sort. These files were called Ebeam fracture format files.

The Ebeam Fracture Format File

The Ebeam Fracture Format (*EFF*) file consists of one physical file which is subdivided into many logical subfiles. Every subfile may be open for writing simultaneously. For the sake of simplicity the following describes the organisation of a one layer (mask level) EFF file. Multiple layer files are achieved by the obvious extension of this organisation.

The EFF file is built on top of the random access, block based IO system[1]. The first blocks are given up to pointers to the start of each stripe file, each pointer being in two parts: a block number and a within block offset. The final word in any data block is a similar pointer to the continuation of this stripe file. See Figure 4–2. Once the fracture has been completed the correctly ordered

[1]IMP has no such system and so one was developed for this thesis

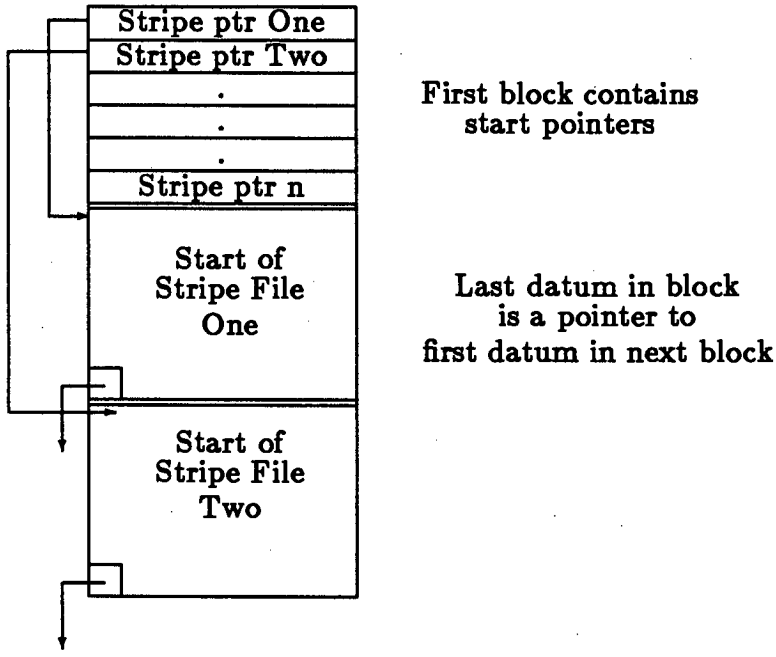


Figure 4–2: Logical construction of the EFF files

pattern tape is generated by reading back the EFF file a stripe file at a time. This operation is completely independent of the fracture processing (and thus can be pipelined and organised to run at the speed of a magnetic tape drive). Currently it is not uncommon for fracturing to be performed in two similar phases, the motivation being to reduce possession time of the tape drive.

Alterations to the Scan Line Algorithm

Obviously the precise algorithm used to perform the decomposition is completely independent of the technique described above to parallelise the process of Ebeam fracture. For the experiments detailed below an adapted scan line algorithm was used. In addition to events at minima and events at the ends of edges two new events are introduced. Whenever a new edge is introduced into the scan line (or an edge is changed by virtue of an end edge event) an intersection check is made with the bounding box of the stripe. If the edge crosses a

boundary at the top a *stripe end* event is introduced and if at the sides a *stripe edge* event.

At a stripe end event trapezia are output at all the edge pairs before processing continues (with trapezia now being output to a new stripe subfile). In the sequential system processing had to terminate at an end edge event, but the *state* of the fracture was saved to enable faster processing when handling the next stripe.

At a stripe edge event two new edges are introduced. These edges run vertically and divide the old edge pair into two new pairs, each of which is in a separate stripe. See Figure 4–3. These two events have the effect of clipping

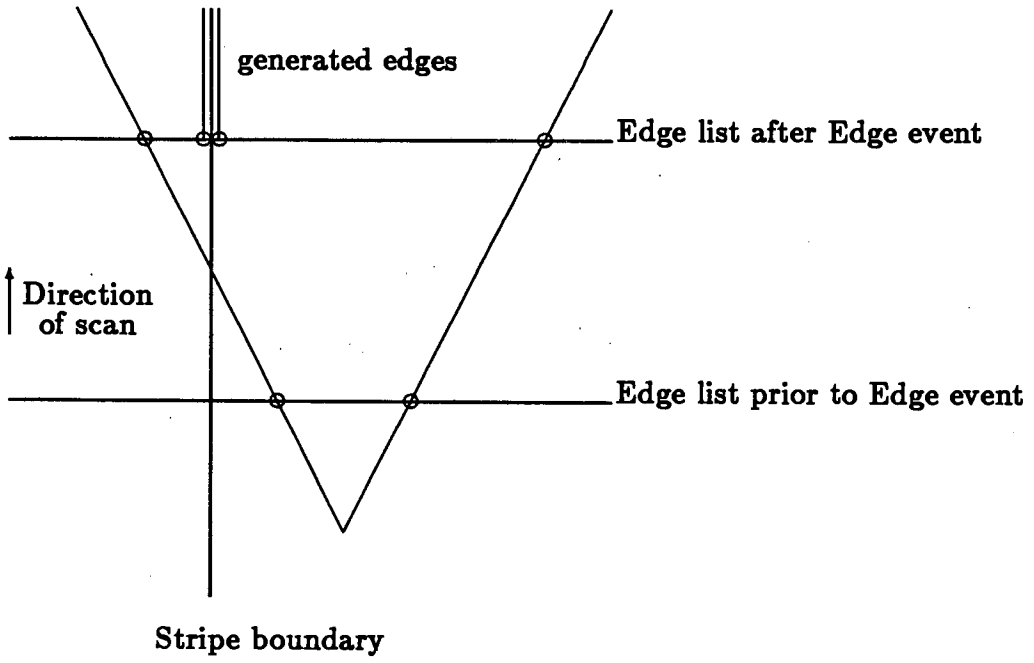


Figure 4–3: Edge Event

any stripe crossing polygons to the stripe boundaries, while ensuring that the data both within and without all stripes is preserved.

4.2.2 The system architecture

The envisaged architecture is similar to that of the merge system. See Figure 4–4. A single *input processor* reads the input data and passes polygons to one of a bank of *fracture processors*, whichever is not currently performing any calculation. Each fracture processor decomposes the polygons and passes the trapezia to the output processor which drives the EFF file. As a totally separate function the EFF file is copied across to the required format of magnetic tape.

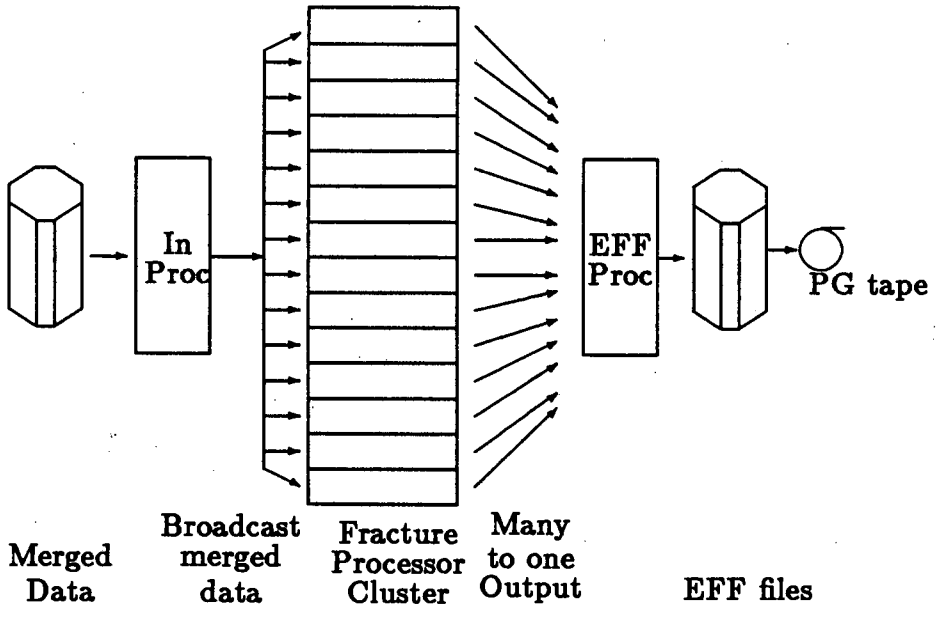


Figure 4–4: System architecture of the Parallel Ebeam fracture

4.2.3 Implementation

Given these techniques for parallelising the Ebeam fracture process it remains only to demonstrate its effectiveness or otherwise and to examine at least the first stages of performance improvements. As for the merge stage this has been achieved by an emulation of the system.

Each fracture processor is emulated by an individual fracture time clock. When a polygon is fractured it is assigned to the fracture time clock with

the minimum time recorded. In addition to the fracture processors, both the input and the output processors are emulated with separate clocks. If the input processor has to wait for a fracture processor to come free or *vice versa* the necessary clocks are resynchronised (ie the clock brought forward, thus emulating a period of null processing).

As for the merge stage, the communication cost was measured by generating a trace file of events. This was normally a series of time-stamps, each with a value, indicating the amount of data transferred, each one being written either when a polygon was passed to a fracture processor, or when a trapezium was passed to the output processor.

4.3 Results and Modifications

Using the emulation enough data was captured on how the system would perform with respect to the major points of interest - accuracy, speed, communication costs, memory requirement. Where performance in any of these fields was less than expected the emulation was used to explore the directions that further development should take. As for the merge stage, several first stage improvements were incorporated.

4.3.1 Accuracy and Flash Count

Since the fracture system works to a level of resolution of 0.01μ, so long as rounding to the Ebeam address units (which are never less than 0.1μ) is performed correctly, accuracy will never be present as a problem to the system.

A useful side-effect of the amended scan line algorithm is that the number of flashes produced is reduced from that generated by a sequential decompose and clip fracture system. When a stripe-spanning polygon is fractured, any event will only affect edge pairs (and thus cause trapezia to be output) within the stripe where they occur. See Figure 4–5.

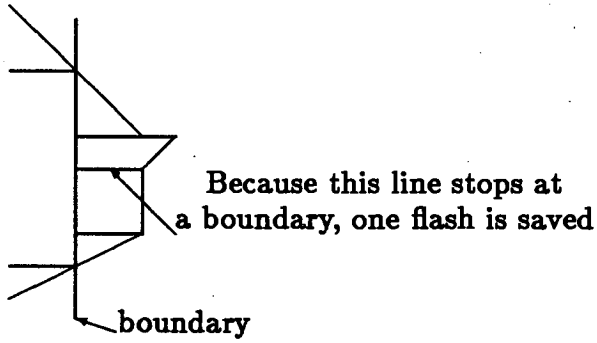


Figure 4–5: Amended Algorithm: The reduced trapezia count

4.3.2 Memory Cost

The cost of memory can be divided up into two distinct parts. Of greatest importance is the memory requirements of the fracture processors. However techniques which control the size of the fracture files generated are also of interest.

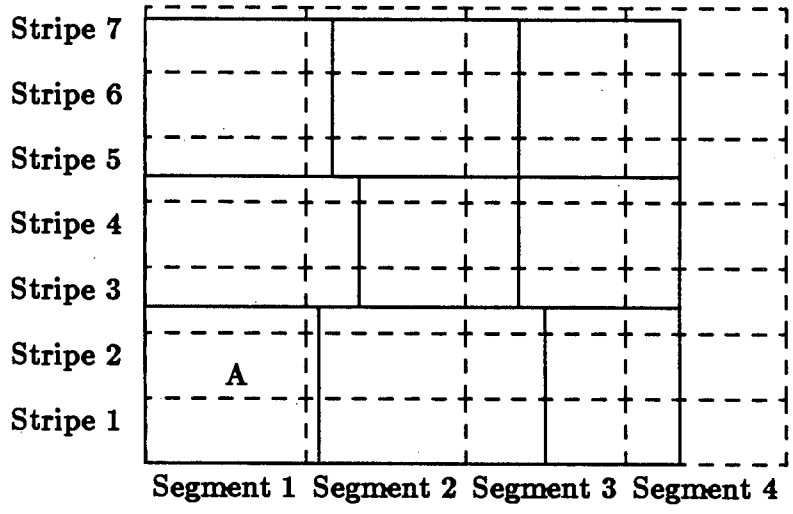
Fracture processor memory

As for the merge stage this is vital, but in sharp distinction to the merge phase the memory requirements for decomposition are minimal since the polygons are fractured individually. The memory requirements for any fracture processor are just the amount needed to store the largest polygon and the code and data-structures needed to decompose it (which will be considerably less than that needed to store the polygon itself). The size of a polygon will never be so vast that even a megabyte of memory is required to store it. Thus fracture processor memory requirements are not a limitation of parallel Ebeam fracture.

Controlling the size of EFF files

The described implementation of EFF files is very wasteful of disk space. On average the free space (*ullage*) at the end of every stripe file will be half a block

in size. The smallest block size is one half of a kilobyte and so wasted space may become several hundred kilobytes. It is clearly advisable to reduce this as much as possible. If any stripe subfile can be closed before the processing has terminated then the ullage in the file's last block can be re-used. A stripe file



When region A has been processed, stripe files 1 and 2 within segment 1 can be closed.

Figure 4–6: Recovering Ullage in EFF Files

can be closed only when all the polygons covering that stripe's area have been processed. This is different from the pre-ordering required by the sequential algorithm. That this has been achieved may be deduced from the intrinsic order of the input.

During the merge stage the output from each processor (the *processor data set*) was kept logically distinct. When the data set for one processor has been completely fractured, all the subfiles for stripes wholly covered by this merge processor (and previously dealt with merge processors) may be closed since no further trapezoids will be output to these files. Any space left at the end of the final block of the subfile may be reused. See Figure 4–6.

The closing of the subfiles and the reclamation of space may take place incrementally during processing of one data set. The merge system ouputs all

polygons in order of decreasing global minimum Y. If the data was read in reverse order the polygons would appear in order of *increasing* global minima. When a polygon has been processed there will be no more data (within the current processor data set) below the lowest point of that polygon. The sub file of any stripe below this minimum Y (and totally enclosed by the merge processor data area) may be immediately closed. Thus wasted space will be reduced to a minimum. This final space saving is particularly useful when only one processor was used to perform the merge.

In order to allow the polygons to be read in reverse order and a data set at a time, the standard output of the merge system was altered somewhat. As for EFF files, this file format is based upon a block based IO system and consists of many sub files (one for each processor data set). Whereas output from the merge stage normally consists of a header followed by the data points, the amended system consists of data points followed by a header. Furthermore the indices held in the headers of the block file point to the end of the data set and the first (not last) word in a block is a continuation pointer to the previous (not next) block. The last data set is that of those shapes which have been recombined. This is the first to be read.

Given that the output stage of the merge system can be the bottleneck on multiprocessors this coding needs to be done exceptionally carefully. Indeed it may well be that the extra overhead to generate these specialised output files is such that the extra space in the EFF files caused by the ullage is acceptable. Furthermore, if the performing of input becomes a bottleneck it would be sensible to remove the extra cost of reading complex input files.

4.3.3 Load Balance

The graphs given in Tables 4–5 and 4–6 give the time taken to fracture one layer of a design. It is obvious from these graphs that an even load balance is not being achieved. The reason for this is that the output from the merge stage of this example consisted of very many polygons of relatively small size and

one very large polygon. That this is a moderately common scheme of things is born out by reference to Table 3–6. The fracture processor takes a very long time to complete, by which time the rest of the data for that layer has been fractured. Thus all but that fracture processor spend the majority of the time idling. Massive polygons also have a detrimental effect upon the bandwidths - the passing of one massive polygon to a fracture processor is liable to swamp the available input bandwidth.

It is therefore advantageous to ensure that the input polygons are not excessively large. As mentioned in the previous chapter there is a special module in the merge stage which achieves this. This *crumble* module was specifically designed to reduce massive polygons to below a threshold set either by plotting software (some plotters cannot handle polygon fill for large polygons) or by a format definition such as Calma GDS-1 (LU) format which allows 117 vertices.

The reduction of vertex count for distributed Ebeam fracture does not present such a tight threshold. All that needs to be achieved is an avoidance of massive polygons. Furthermore using the crumble routine will add to the time taken to do the merge. Thus crumbling is best avoided if it can be.

If the recombination STITCH phase of the merge process is removed, then the vertex count of the largest polygons will naturally decrease (since large polygons will tend to be spread over more than one merge area). Since Ebeam fracture data can represent acute angles there is no possible loss of accuracy introduced by non-Manhattan wires crossing boundaries. The only disadvantage is an increase in the number of trapezia output.

The graphs given in Tables 4–7 and 4–8 indicate that the removal of the STITCH module does indeed give a much more balanced load. This removal has another advantage. The recombination stage of the merge process takes place in the sequential part of the system and, as can be seen in Table 3–8 for large designs this can take a large proportion of the overall time.

It is immediately apparent from the Tables 4–7 and 4–8 that once balance has been achieved the input and output processors become the bottlenecks and

as such the targets for careful coding. It is particularly worth pointing out that the IO package used in the emulations was not a 'professional product' but was just another part of the necessary programming required in the research for this thesis. Experience shows that the performance of this system could probably be improved considerably. An indication of the performance improvements possible in this field is given in Table 4–1.

Old s	New s	Increase percent
412.1	382.1	7.3
426.3	394.7	7.4
427.7	399.7	6.5

Table 4–1: Time taken by Output processor

The only difference between the 'old' and 'new' systems is that four lines of code were migrated from being performed by the output processor to the fracture processor. The cost of doing this is that the bandwidth has to be increased.

In addition to changes made by moving the function from being performed sequentially to being performed in parallel, there are other 'traditional' software engineering techniques for speeding up the output and input processor, some of which were described in Chapter 1.

4.3.4 Processor Numbers

The distributed fracture algorithm can be divided into three parts - input, output and decomposition. For any given input data the time taken to input it and to output the trapezia should remain constant, regardless of the number of fracture processors. Similarly the *total fracture time*, defined as being the

sum of the time that all the fracture processors spend decomposing polygons *not* idling, should be constant. This is indicated by Table 4–2.

Number of Processors	Input Time	Total Fracture Time	Output Time
1	153.6	987.5	388.7
2	153.8	990.5	399.5
3	151.7	1000.0	399.9
4	154.3	996.6	392.1
5	153.6	998.3	385.9
6	156.9	996.0	385.6
7	154.4	990.8	399.7
8	153.7	965.3	392.1
9	152.5	982.7	394.5
10	153.5	984.2	384.3
11	162.1	996.8	388.3
12	153.3	985.7	383.1

Table 4–2: Breakdown of fracture timings for one layer

It is desirable to choose a value for the number of fracture processors so as to avoid processors being unneccesarily idle. The time taken by the fracture processors overall (ie the time taken by the slowest processor) should be as close as possible to time taken by the slowest of the input or output processor. Thus the number of fracture processors N_F can be calculated as:

$$N_F = \lceil \frac{T_{F*}}{\max(T_i, T_o)} \rceil$$

Where T_{F*} is the total fracture time, T_i is the input time and T_o is the output time. For the data given in Table 4–2, N_F is 3.

4.3.5 Emulated Speed

Assume that a good load balance is achieved and that communication costs are negligible. Let T_n be the time taken for a parallel system with n fracture processors and T_0 be the time taken by the equivalent sequential system. T_{Fn} is the time taken to perform fracture by the nth processor.

$$\begin{aligned} T_0 &= T_i + T_o + T_{F*} \\ T_n &= \max(T_i, T_o, T_{Fn}) \\ &= T_o \text{ when } n = N_F \end{aligned}$$

The speedup is given as

$$\begin{aligned} speedup_n &= \frac{T_0}{T_n} \\ &= \frac{T_i}{T_o} + 1 + \frac{T_{F*}}{T_o} \end{aligned}$$

Since $0 \leq \frac{T_i}{T_o} \leq 1$ and $N_F \leq \frac{T_{F*}}{T_o} \leq N_F + 1$

$$1 + N_F \leq speedup_n \leq 3 + N_F$$

So, allowing for communication cost and non perfect load balancing, it should be reasonable to assume a speedup of at least 3 times *for this particular setup.* As mentioned above, a more skilfully implemented IO package would allow immediate and massive improvments to this. Table 4–3 gives details of the time taken and factor speedup. All times are given in seconds. As can be seen an order of 3 times speedup has been achieved regardless of the size of the problem.

It should be pointed out that the calculations upon which this section is drawn were all based on designs which had a large amount of non-Manhattan geometry in them

4.3.6 Communication Costs

As for the merge stage, the cost of communication was measured by generating a trace file to which a time-stamp datum was appended whenever any commu-

Total Time	Emulated Time	Fractional Speedup
85.6	23.0	3.7
236.2	63.8	3.7
555.9	140.5	3.9
1.7	0.46	3.7
252.0	60.6	4.2
17.9	5.1	3.5
411.6	87.3	4.7

Table 4–3: Ebeam Fracture - Times and speedup

nication was generated. From this both the average and peak bandwidth could be calculated. The average bandwidth was calculated as the amount of data output (or input) divided by the amount of time over which input or output occurred. This gives a rough idea of the loading which a communications channel would be expected to handle. The peak bandwidth is simply calculated as the maximum number of bits moved over a discrete time-step (in this case one hundredth of a second) divided by that time-step. Table 4–4 gives details.

As can be seen both the average and peak 'bandwidth' measurements are somewhat different for each experiment (these are the same fractures as described in Table 4–3). However the 'average bandwidth' is well below one megabit per second and so should not present any problems in terms of hardware implementation. The gross variation in peak measurement - which also shows up as great variation on the graph in Table 4–9 is a function both of the small timestep used in the analysis and the fact that the input contained one massive polygon.

It is interesting to note that the input bandwidth is always larger than the output. This can be attributed to the difference in the coding used for the input and the output data. Consider for instance a box. As input this might be

Input bandwidth		Output bandwidth	
Peak	Ave.	Peak	Ave.
Mb/s	kb/s	Mb/s	kb/s
0.32	96.5	5.12	52
0.32	100.0	5.12	53
57.2	592.0	916	51.4
0.12	56.6	1.84	30.7
24.1	324.0	385	57.5
0.17	72.7	2.66	36.2
1.31	149.5	20.9	68.4

Table 4–4: Ebeam fracture. Average and peak bandwidth

represented by its bounding box. Since the box may be of any size and anywhere on the mask each coordinate would need to be stored as a four byte integer. Thus for input a box requires 16 bytes to be represented and this does not allow for the box descriptor. On output it is known that all output will be trapezia and will be within stripes. Thus the output could consist of a stripe identifier, a segment identifier, a bottom and top Y coordinate and 4 X coordinates. Depending on the stripe height and segment width this can be represented as less than 16 bytes[2]. There are obviously greater data compressions which can be achieved. It should be remembered that greater speedup is bound to have an adverse impact on the bandwidth.

[2]for VARIAN ALF format this is 15 bytes

4.4 Conclusions

As for the merge stage a parallel algorithm has been demonstrated. Again the limitation on the amount of available parallelism is not due to a failing of the general technique, but because of poor implementation of time-critical parts of the system. Even had more time been available it was not within the bounds of this research to follow up the tail-chasing path of further improvement, but a critical examination and reimplementation of the input and especially the output systems would be the most important next stage of development. It should be stressed that since the Ebeam fracture stage is less CPU intensive than the merge stage it is less critical to achieve great improvements.

Another development which could be investigated is the post processing of the EFF files such that the compacted format which some Ebeam machines accept could be generated. This has been done (with surprisingly good results) for a sequential system. Work would need to be carried out to see how this might be adapted to become part of a parallel system.

Similarily again to the merge stage the techniques used to parallelise the fracture stage are not all dependent upon which algorithm is used to perform the decomposition of the polygons. The algorithm used was chosen for the pragmatic reason that it was easy to implement. A more complex algorithm which gave better (in the sense of being closer to optimal) results could easily be adopted. Adoption of a different decomposition algorithm would of course require N_F to be recalculated. Algorithms with considerably worse asymptotic time performance might need more careful load balancing.

In sharp contrast with the merge stage, the parallel system for Ebeam fracture does not become amenable to greater levels of parallelism as the data size becomes larger. The optimal size for a fracture system is fixed (once the algorithms and their implementations have been fixed). Thus even for very small designs the speedup due to parallelism is achievable. See Table 4–3.

As for merge there is another level of parallelism available over and above that demonstrated in this chapter - the possibilities of handling more than one layer at a time in parallel. This is particularly useful for the Ebeam fracture stage where the number of processors used is rather less than for the merge stage.

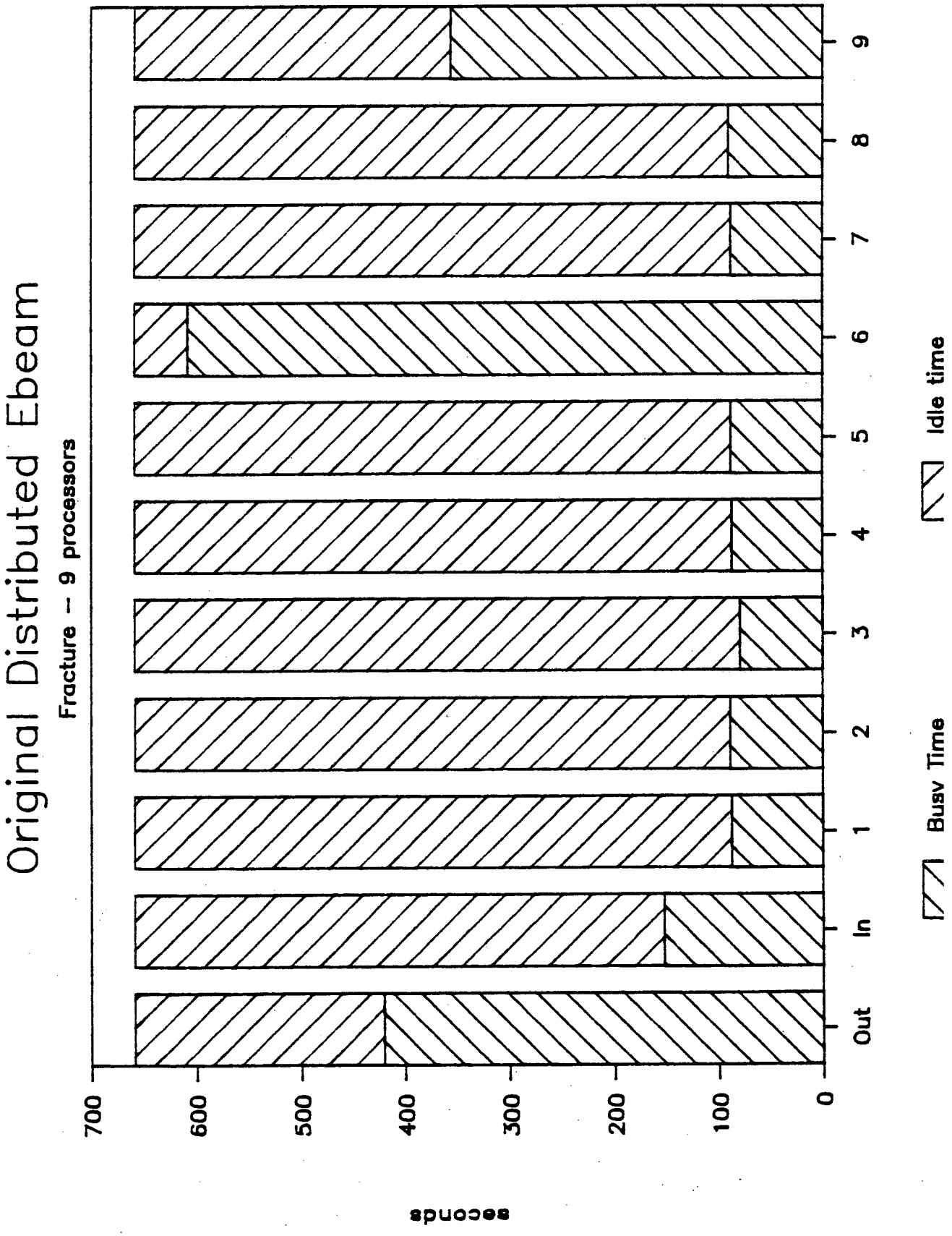


Table 4–5: Original distributed Fracture for 9 processors

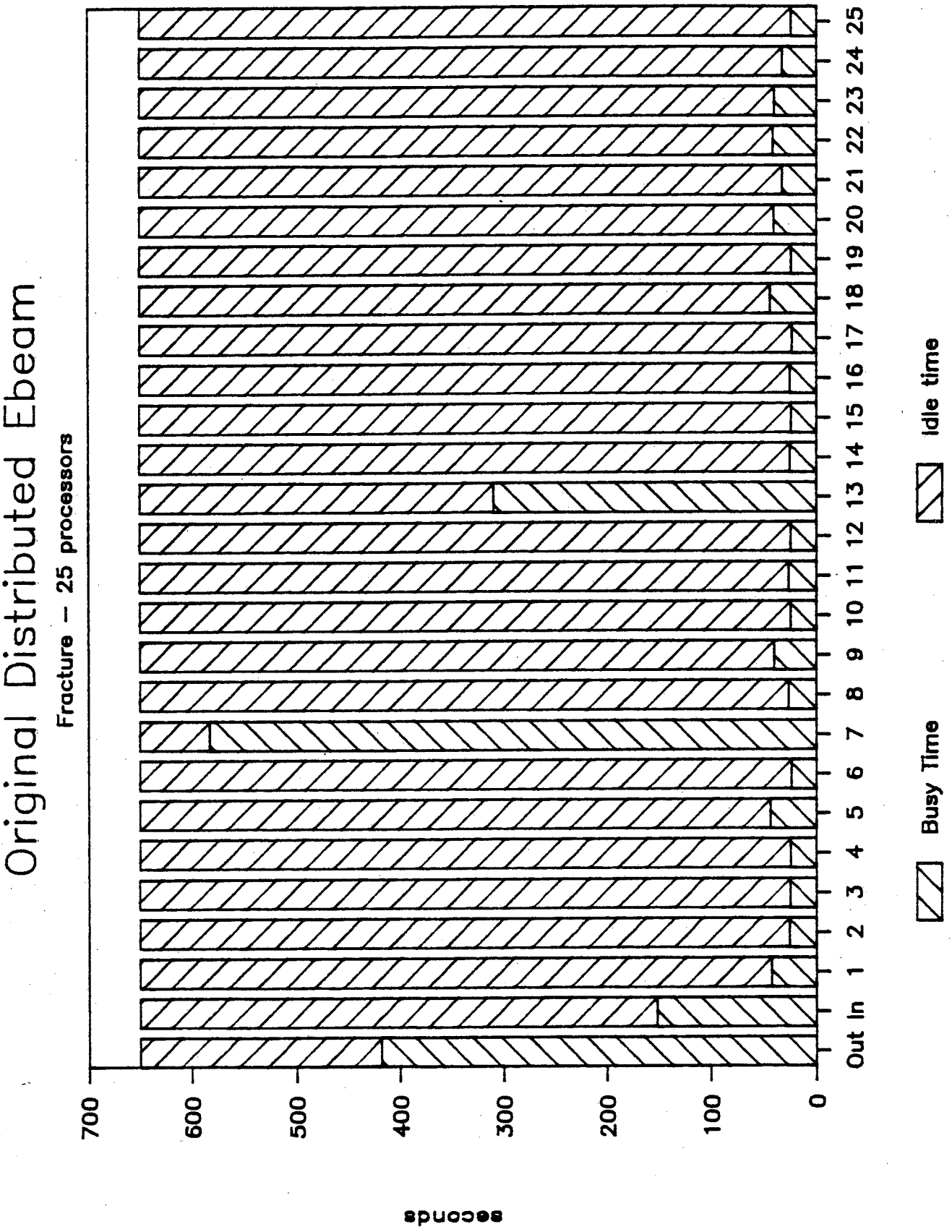


Table 4–6: Original distributed Fracture for 25 processors

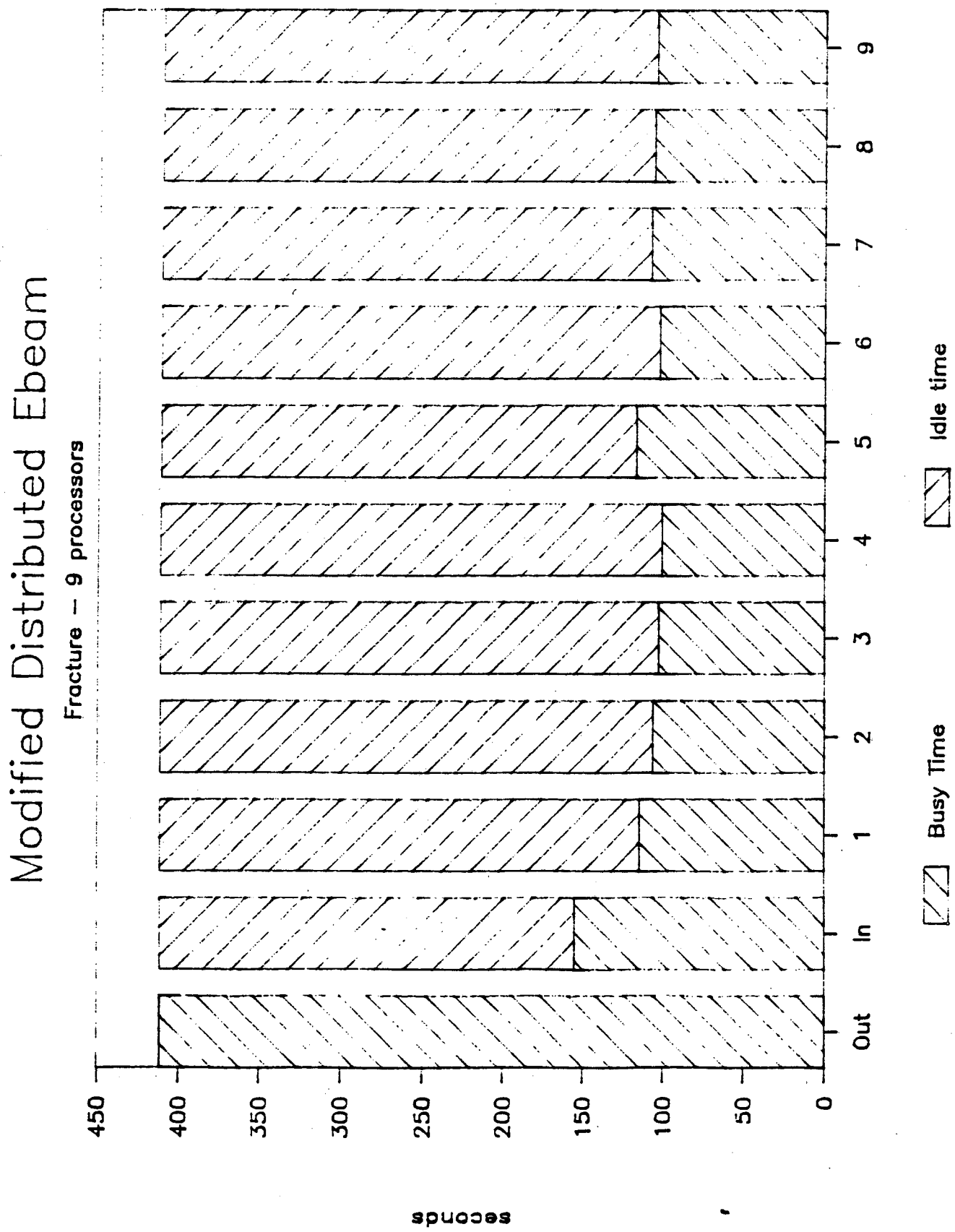


Table 4–7: Amended distributed Fracture for 9 processors

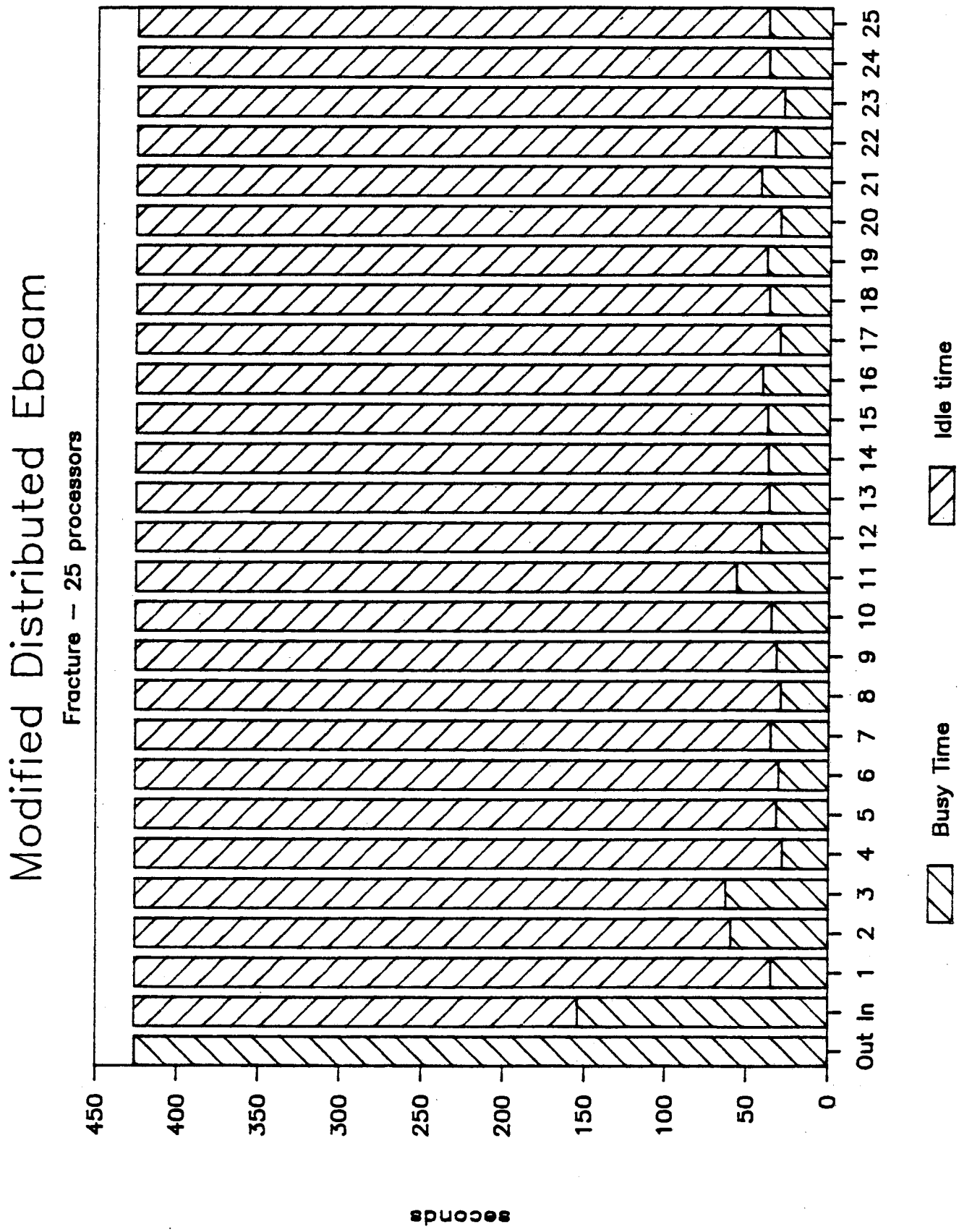


Table 4–8: Amended distributed Fracture for 25 processors

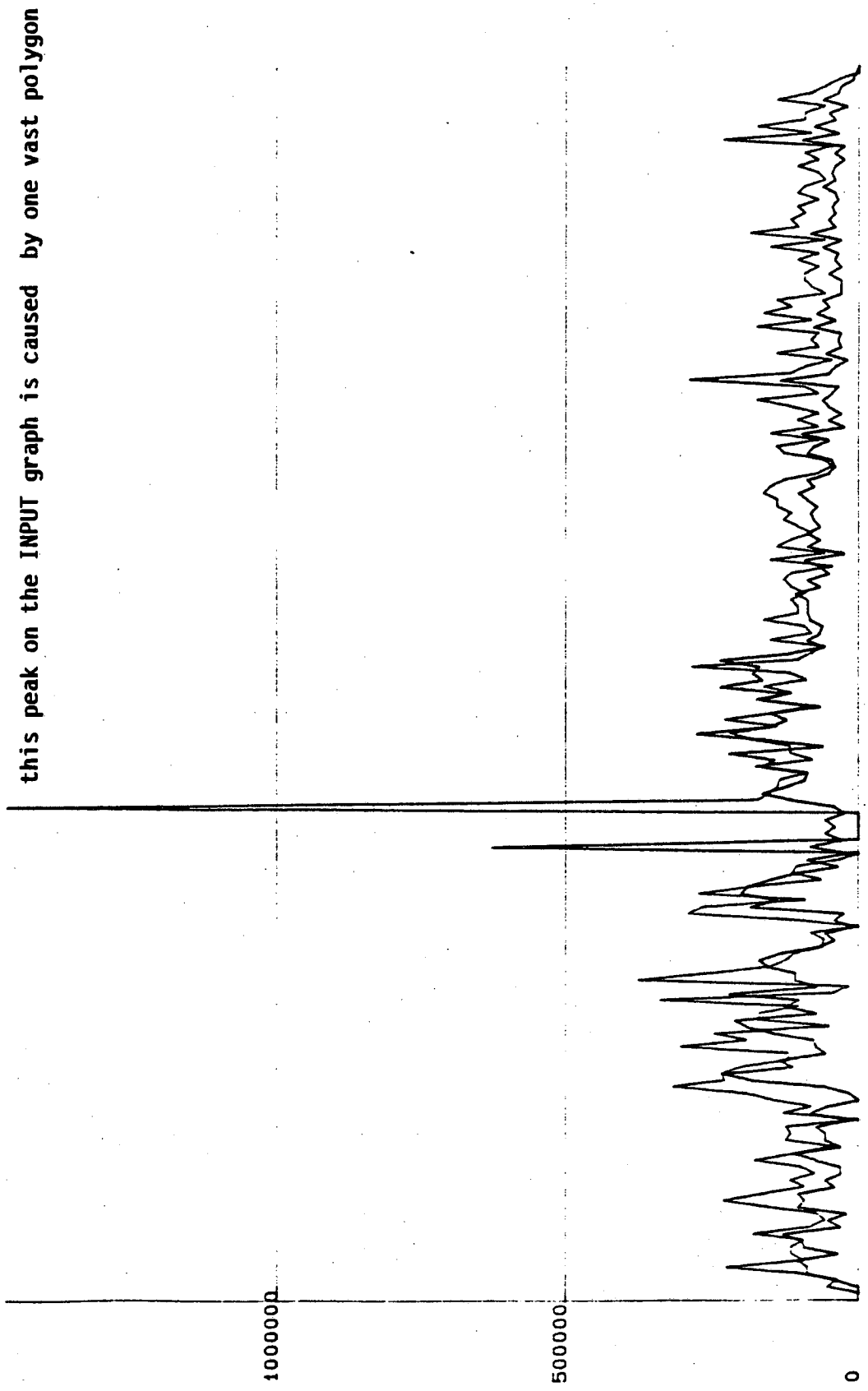


Table 4.9 Ebeam fracture . Data rate against time.

Chapter 5

The Optical Fracture Stage

As for Ebeam fracture, optical fracture consists of two stages - the decomposition of the polygons and their sorting. Depending on the precise algorithm used in the decomposition and the make up of the input data the time taken to fracture for optical machines varies enormously. It is therefore advantageous to investigate methods of speeding optical fracture, especially in the cases where it runs particularly slowly.

5.1 Algorithms for Ebeam Fracture

Again similarily to Ebeam fracture, study of algorithms for optical fracture consists of two parts, investigation of the algorithms which perform the decomposition and of those algorithms which sort the output.

5.1.1 Theoretical Considerations in Decomposition

Decomposition of a polygon can either be by way of a *covering* or a *partitioning*, depending upon whether overlap is allowed or not in the output. For Ebeam fracture a partitioning was used, but for optical fracture, in order to reduce the flash count (and in many cases in order to make the decomposition possible), a covering is usually adopted. Unfortunately minimal cover even for Manhattan

polygons is *NP* complete [J82]. Thus approximation techniques have to be adopted.

The order in which the flashes are written to the pattern tape is the order in which they will be exposed on the plate. This critically affects the speed at which the plate is exposed. To further complicate the issue, the speed of the various mechanical functions (rotation, movement and so forth) varies, not only between machine types but also *between machines of the same type.* The problem of finding the best order for the flashes is thus equivalent to the Travelling Salesman Problem and as such is *NP* complete. Thus approximation techniques or heuristics have to be adopted to achieve a reasonable flash rate within a reasonable period of time.

5.1.2 Algorithms for Polygon Decomposition

The covering of non-Manhattan polygons, although a thing which humans can do with consummate ease, is difficult to express algorithmically. Indeed very early pattern generation software would only fracture all angle polygons which were expressed, not as vertex lists, but in terms of high level constructs - for instance arcs, circles and rectangles. This already complex problem is further complicated by the fact that in many cases a lessened flash count does not necessarily mean a better fracture. Consider, for example, the highly complicated polygon shown in Figure 5–1. Figure 5–2 is a fracture with 265 flashes and Figure 5–3 has 444 flashes. Although Figure 5–2 has less flashes the amount of overlapping is such that it would be impossible to generate successful reticles from this data.

In the absence of polynomial time complexity bounds on the problem, most implementations and algorithms developed for covering polygons for use with optical pattern generation equipment have been ad-hoc in nature. One technique which seems reasonably common is for optical fracture systems to have two distinct decomposition algorithms: one for the decomposition of Manhattan polygons and one for the decomposition non-Manhattan polygons. [Ber86] and

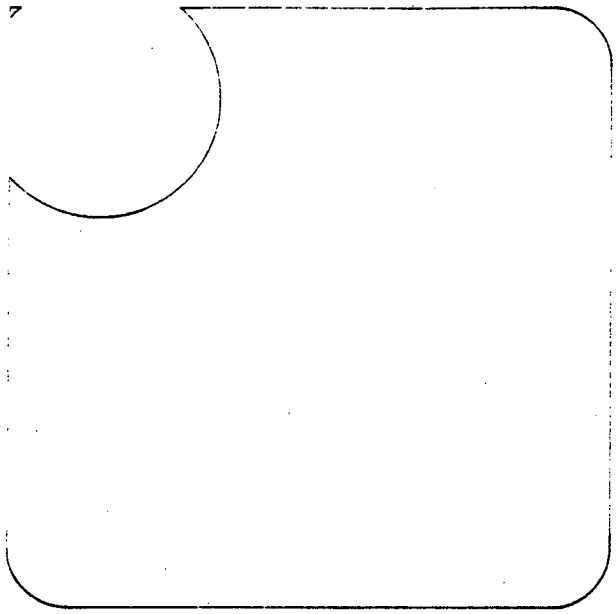


Figure 5–1: A complicated Polygo

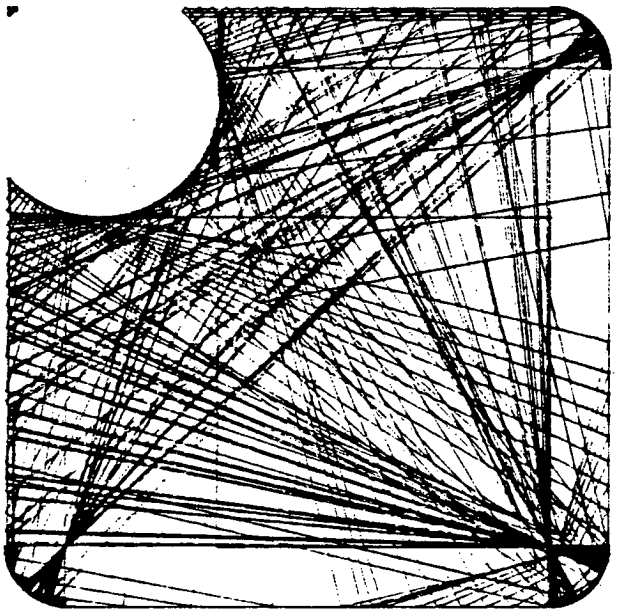


Figure 5–2: A Possible Covering with 265 Flashes

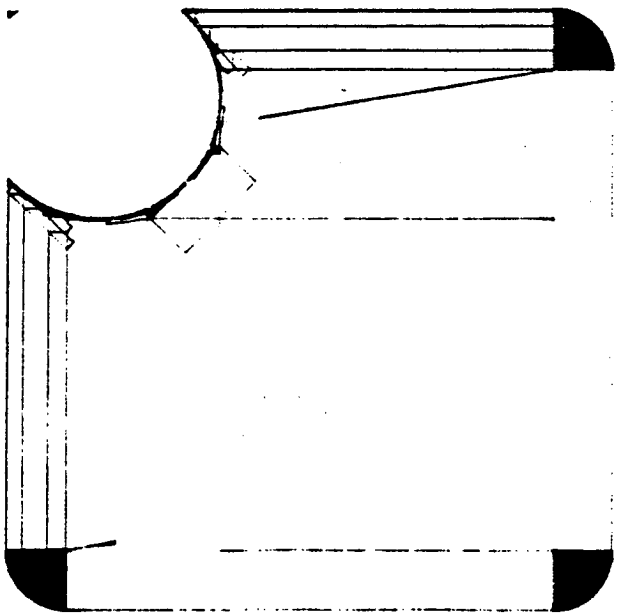


Figure 5–3: A Better Covering with 444 Flashes

[Heg82] describe all angle fracture programs of the 'ad-hoc' type. These work by successively growing rectangles out from the edges of the input polygon until the whole polygon is covered. The outward growth is limited by the 'other' edge either hitting another edge of the polygon, or by it completely covering a *frontier*, that is an internal edge of a previously flashed polygon. Heuristics are used to control the algorithm, these include which edge to grow from next, whether to extend the edge sideways (if possible) prior to growing the rectangle and whether to stop at a frontier or not. These heuristics have a great impact on the output and indeed the performance. For instance the difference between Figure 5-2 and 5-3 was made solely by adjusting these heuristics. Further investigation of these is continuing, but is outwith the scope of this thesis.

[GG83] describes the PTR (polygon to rectangle) algorithm. This algorithm, not unrelated to the scan line algorithms used in the last two chapters, *partitions* a Manhattan polygon into Manhattan rectangles. In some cases, the use of non-Manhattan flashes can reduce the flash count. This is a highly infrequent occurrence and the added complexity of searching for such an occurence makes its inclusion self-defeating. The work described in this chapter was based upon the all angle decomposition algorithm described in [Ber86] and an adaptation of the PTR algorithm which generates a covering of the input polygons.

5.1.3 Sorting

In the system which was the basis for the work described in this chapter, the *NP* complete problem of finding the optimal flash ordering was handled, by applying heuristics. Every flash, as it is generated, is sorted, first by angle of rotation, then by Y position and finally by X position. When all the polygons which make up one layer have been decomposed (or a flash count threshold has been reached), the sort tree is unwound in such a way that the rotation changes least frequently, then the Y value and finally the X. The output is such that the plate moves back and forward in the X direction as the Y increases thus the overhead of fly-back is avoided. This ***boustrophedonal*** method of sorting

the output flashes produces reasonable results and is particularly effective on machines which can 'flash on the fly'.

5.2 Parallelising Optical Fracture

As for Ebeam fracture, the polygon decomposition, represents an obvious choice of unit for parallelisation. A major difference between the parallelising of the optical fracture and that for Ebeam was that whereas the parallel Ebeam fracture system was based upon theoretically sound algorithms, which had little scope for improvement by software engineering techniques, the Optical fracture system, in particular the decomposition part, was at a less well developed stage. Thus the basic system could be improved by standard software engineering practices without resorting to use of parallelism. The results of this chapter show that algorithmically sound algorithms are not a necessary basis for performance improvement by use of concurrency. It is, however, usually preferable to improve the implementation by established software engineering methods before launching into the fray with parallelism.

5.2.1 Parallel Optical Fracture

If performance of the decomposition is the most time consuming part of performing a fracture (as it is in the time dependent cases in particular), the obvious method of parallelising optical fracture is by development of the system described in the previous chapter for the fracturing for Ebeam machines.

Since the sort takes place after the decomposition polygons may be decomposed in any order; in particular they can be processed in the order in which the merge stage generates them. With the experience gained when parallelising the merge and Ebeam fracture stages it is obvious that a method of parallelising fracture is for each polygon to be assigned to fracture processors as they come free with a dedicated output processor serving to order the rectangles prior

to writing the PG tape. Rectangles, of course, pass immediately from input to output with no intermediate decomposition stage, being merely sorted and translated from input to output format.

In contrast to the Ebeam fracture stage, correct functioning of the system does not rely on non-standard input or output - in fact the standard input and ouput were used in the emulation described below.

5.2.2 System architecture

The architecture of the resulting system is very similar to that for the Ebeam fracture system which itself was not dissimilar to the merge system. The main changes are that the output can go directly to magnetic tape (although buffering through disk intermediate files to reduce tape drive possession time is still useful) and the fact that some shapes (rectangles) are immediately passed to output. See Figure 5–4

5.2.3 Implementation

Again, experiments were performed on the system by means of an emulation of the system. Just as the parallel optical fracture system was based upon that developed for the Ebeam, so was the emulation with each processor, whether decomposition, input or output, being associated with an emulating clock. Procedures were provide to emulate synchronisation and allowed selection of 'next free processor'. Obviously the overall emulated time is the greatest of those shown on the emulating clocks, when processing has terminated.

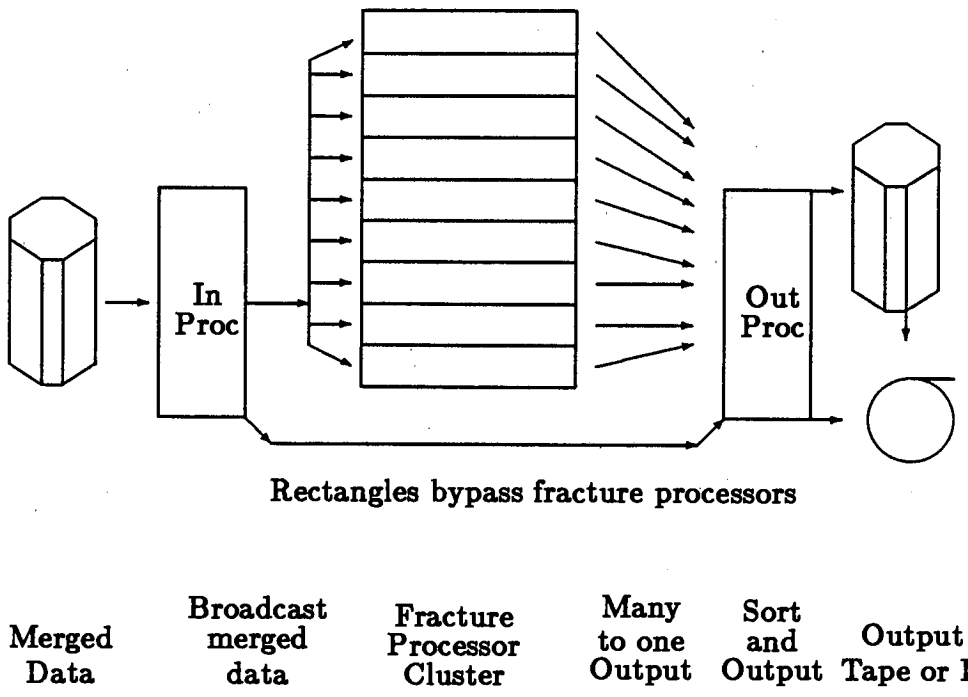


Figure 5–4: System architecture for Parallel Optical Fracture

5.3 Results and Modifications

The emulated parallel system was used as the basis for experiments to evaluate the performance of the system. Some improvements were included during these experiments, and these are described in the relevant sections.

The same time-stamping techniques used in Chapters 3 and 4 were used to measure the overheads associated with communication cost in a manner amenable to off line analysis.

5.3.1 Accuracy

Since the basic decomposition and sorting algorithms are unchanged there will be no change to the output data. Thus parallelising the system introduces no

further inaccuracies upon the standard system. However the standard system itself is less accurate than might be desired. In particular a fundamental design decision with the original system was that all geometry should be covered. Although this appears to be a sensible approach there are cases when it can badly affect the quality of the fractured data - Figure 5–5 illustrates.

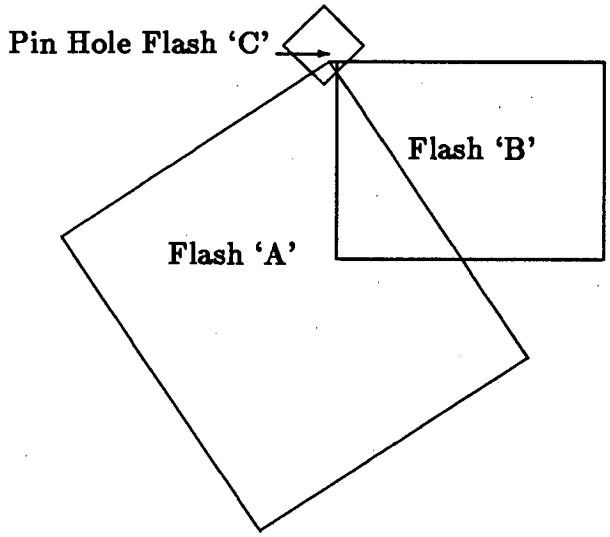


Figure 5–5: The pinhole accuracy problem

The two flashes 'A' and 'B' leave a minute 'pin-hole' in the original data. If this flash is ignored the final mask quality would not be affected. Since the algorithm seeks to cover all the data and will not terminate until such a time as this has been done a third flash 'C' is generated. This flash (although of the minimum size generatable by the machine) is much larger than the hole it is covering and furthermore is offset so that it appears as a bulge on the edge of the field. In such cases an algorithm which does not seek to cover all the data might be favoured.

The methods for handling this sort of inaccuracy are not part of the work described here. It is the subject of current and future research. It is included here only for the sake of completeness and as a reminder that any parallel system developed by the techniques described in this thesis is only going to be as good as the system upon which it is based.

5.3.2 Memory Cost

As for Ebeam fracture, memory usage of the system is not great. Indeed the limiting factor is the size of the search tree used to sort the output flashes. This can be controlled by governing the threshold at which the sort trees are purged.

5.3.3 Load Balance

The graph in Table 5–6 shows the idle and the busy time for the fracture of one layer of a complex, predominantly non-Manhattan design. The fracture was carried out on an emulated system consisting of an input processor twelve fracture processors and an output processor which was also used to perform the sorting. Quite obviously for this example at least, an even load balance has been achieved.

A good load balance is achieved. The most important reason for this is the fact that the size of the input polygons is limited. The basic decomposition algorithm for non-Manhattan geometry is of such an algorithmic complexity that large input polygons are exorbitantly costly to decompose. Thus it is normal with this system to limit the size of the input polygons. Had this not been done it is very likely that the system would show a very bad load balance - considerably worse than that shown for the Ebeam decomposition case in the previous chapter, where the computational complexity of the decomposition stage is much lower.

5.3.4 Processor Numbers

It is reasonable to assume that the total time taken to perform input, decomposition and sorting and output, will remain constant regardless of the number of processors performing the decomposition. This is confirmed by inspection of Table 5–1, which shows, for three separate emulated systems, the amount of time taken in performance of these three functions for each of three layers taken from two designs.

Layer	Polygon Count	Processor Count	Input Time(s)	Decomp. Time (s)	Output Time(s)	Flash Count
Active †	3,253	4	53.4	2,204	121	15,844
		8	55.0	2,231	123	
		12	54.4	2,188	120	
Contact †	9,338	4	140	1,150	110	16,994
		8	143	1,179	113	
		12	143	1,173	112	
Diff. ‡	107,692	4	1,162	1,938	4,770	287,818
		8	1,217	1,970	4,723	
		12	1,190	1,902	4,708	

† indicates that the geometry was predominantly non-Manhattan

‡ indicates that the geometry was predominantly Manhattan

Table 5–1: Optical fracture. Cost breakup per function

Although the time taken per function *does* remain constant regardless of the configuration of the system, the time which decomposition and output take is very variable and completely unpredictable. Obviously the input time varies as a function of the size of the input and the output time on the number of flashes; but there is no way of judging the latter from the former. The time taken to perform the decomposition is equally variable being in particular a function of the makeup of the geometry (note the effect that mainly Manhattan geometry has). In the absence of any obvious trend it is therefore difficult, if not impossible, to make a decision as to an optimal number of processors. Given the figures in Table 5–1, in the case of non-Manhattan geometry as many as possible up to a limit of around 20 (for this configuration) seems feasible. When fracturing Manhattan data even a configuration with one fracture processor might lead to idle time on that processor.

If some of the load on the output or input processor could be migrated to the fracture processors then a greater level of parallelism would be possible. This

is obviously useful in the case of Manhattan input. In all cases the overall time taken to perform whatever function was migrated would of course decrease.

A great deal of the time taken to output, especially in the case of the Manhattan geometry, is due to the sorting of the output flashes. If each processor sorted those flashes which it generated, and a proportion of all the rectangles which originally were passed directly to the output geometry, all that would remain for the output processor to do is merge the (already sorted) shapes and perform the actual output. Doing this means a of change method in which the output communication happens. All the communication to the output processor takes place once all the decomposition has finished (rather than incrementally as previously). Thus the bandwidth required will be greater. The precise level of this bandwidth is discussed later.

The changes outlined above were incorporated in the system. The large, Manhattan, diffusion layer given in Table 5–1 was fractured at various emulated configurations with the result shown in Table 5–2.

As the table shows the load on the output processor has been considerably reduced. The expected similarity in the times taken both to perform the decomposition and sorting and the output are not so marked. This can immediately be traced to the performance of the sorting as the level of distribution varies. In particular the time taken, both to sort and to output peaks is maximum where at least one sort tree needed to be flushed during the performance of the fracture.

Since output is performed *after* the decomposition stage, the best configuration for the multiprocessor fracture of *Manhattan* geometry by this system should be one where the total time taken by the fracture processors is the same as that taken by the input processor. Consulting the data in Table 5–2 shows that this is at $\approx \frac{5829}{929} \approx 6$.

5.3.5 Emulated Speed

Table 5–3 gives the emulated time and the fractional speed up for the same batch of experiments which made up Table 5–2. The fractional speed up is taken against the uniprocessor speed of 2 hours 11 minutes and 13 seconds.

Again the configuration where the sort tree is not output affects the emulated time; but the emulated time is still considerably higher than might reasonably be expected. The reason for this is the restrictiveness of the model of synchronisation implemented by the emulating system. In this model no fracture processor can recieve data from the input processor until the input processor becomes free; furthermore the input processor itself often needs to wait for fracture processors to become free. This equates to there being no input buffering in the fracture processors. A further experiment emulated the same configurations running on the same input data but with no notice being taken of synchronisation (this equating to an infinite buffer being available). The results are shown in Table 5–4.

As can be readily seen, the fractional speedup is considerably increased in this case until all the ouput has to be performed at the end (no intermediate flushing of buffers). This now becomes the new bottleneck and the target for further improvement. In reality the emulated time and thus the fractional speedup will be somewhere between these two extremes, depending on the amount of buffering and the quality of the asynchronous input/output handling code.

5.3.6 Communication Costs

Again, these were measured by generating a time-stamp file during an emulation. These were analysed at a later stage to find the amount of data which needed to be transferred between the input and output processors and the fracture processors. By analysing the occurrance of the time-stamps over (emulated) time the peak rate of transference can be analysed. A simple division gives the overall bandwidth; in implementation terms this means having an infi-

nite buffer. Table 5–5 gives the details of the costs thus derived for emulations of configurations of 4, 6 and 12 processors. The 'strong' synchronisation model described above was used. When calculating the bandwidth the assumption was made that the amount of data comprising a polygon was (8 × the number of points) bytes and that each flash was made up as 12 bytes (4 bytes each for X and Y, 12 bits for rotation and 10 bits each for length and width). A large amount of data compression is quite feasible and so these figures should be considered as worst case. A half second 'snap shot' is used to evaluate maximum bandwidth. All bandwidth values are given in kilobits per second.

The bandwidth values, although high, are not excessive. Furthermore, the peak values are not considerably larger than the average values. This argues that the input and output occur at a fairly balanced rate. This is born out by the graph shown in Table 5–7 which shows the level of input or output activity over time for the 6 processor configuration.

5.4 Conclusions

One of the basic tenets given in the first chapter was to ensure that the basic algorithm was sound prior to embarking on parallelising it. Design of algorithms for decomposition for optical machines could form the basis of a thesis in itself. However although the original optical fracture system was distinctly sub-optimal in the algorithmic sense, the same techniques used to parallelise the other parts of the Pattern Generation system could be applied to it without difficulty to give usable speedup - especially in the particularly time-critical fracture of non-Manhattan geometry. As for Ebeam fracture the speedup was not dependent upon the data being large. Very importantly, when algorithmically superior systems are developed, the techniques developed here could immediately be applied since, as for the preceding chapters, not the algorithm for solving the problem but the *form of the problem* has been the basis of the work.

The critical bottleneck, which cause further speed decrease are the input and output; however since standard input and output systems were used in writing the emulator their impact was not as great as it was for the Ebeam system. Certainly with good coding their impact could be further reduced to allow even greater levels of parallelism.

It would, however, be imprudent to reembark immediately upon a new parallel implementation of this system without first studying in greater depth the precise algorithms used to decompose and sort the data. Certainly great improvements are possible for the sorting system. Work carried out by other people while this thesis was being written has generated improvements of the basic decomposition algorithm resulting in the improvements shown between Figure 5–2 and 5–3, in removal of pinhole flashes (Figure 5–5), and reduction in the time taken to decompose non-Manhattan geometry. Certainly the modules which perform the sorting would benefit from similar attention.

Again, it must be emphasised that no matter how bad the original implementation, the techniques used in this chapter were still usable for improving the speed of any implementation (in this case an optical fracture system). However any problems in the original system will also be present in any resulting parallel system and so it would normally be the case that the original implementation should be as good as possible prior to parallelising it.

Processor Count †	Input Time(s)	Decomposition Time (s) *	Output Time(s)
1	924	5,515	1,650
2	948	5,883	1,697
3	925	6,356	1,681
4	936	6,286	1,692
5	936	7,049 ‡	1,735
6	934	5,840	1,632
7	930	5,748	1,659
8	923	5,620	1,674
9	922	5,523	1,659
10	920	5,453	1,654
11	925	5,368	1,666
12	931	5,315	1,694
AVE	929	5,829	1,679

* decomposition includes some of the sorting costs

† Processor count refers to the number of *fracture processors*

‡ This was the last configuration which involved flushing the search tree

Table 5–2: Modified Optical fracture. Costs per function

Fracture Processors	Emulated Time (s)	Fractional Speed up
1	5,941	1.32
2	3,692	2.13
3	3,150	2.50
4	2,412	3.26
5	1,995	3.94
6	2,984	2.64
7	2,897	2.71
8	2,828	2.78
9	2,759	2.85
10	2,713	2.90
11	2,697	2.92
12	2,705	2.91

Table 5–3: Emulated time and fractional speed-up

Fracture Processors	Emulated Time (s)	Fractional Speed up
1	5,499	1.43
2	2,938	2.67
3	2,116	3.71
4	1,574	5.00
5	1,388	5.67
6	2,609	3.01
7	2,619	3.00
8	2,581	3.05
9	2,639	2.98

Table 5–4: Emulated time and fractional speed-up. 'Weak' synchronisation

Fracture	Input Bandwidth		Output Bandwidth	
Processors	Average	Peak	Average	Peak
4	227	659	16	62
6	266	723	17	18
12	327	851	16	18

Table 5–5: Bandwidth requirements for optical fracture

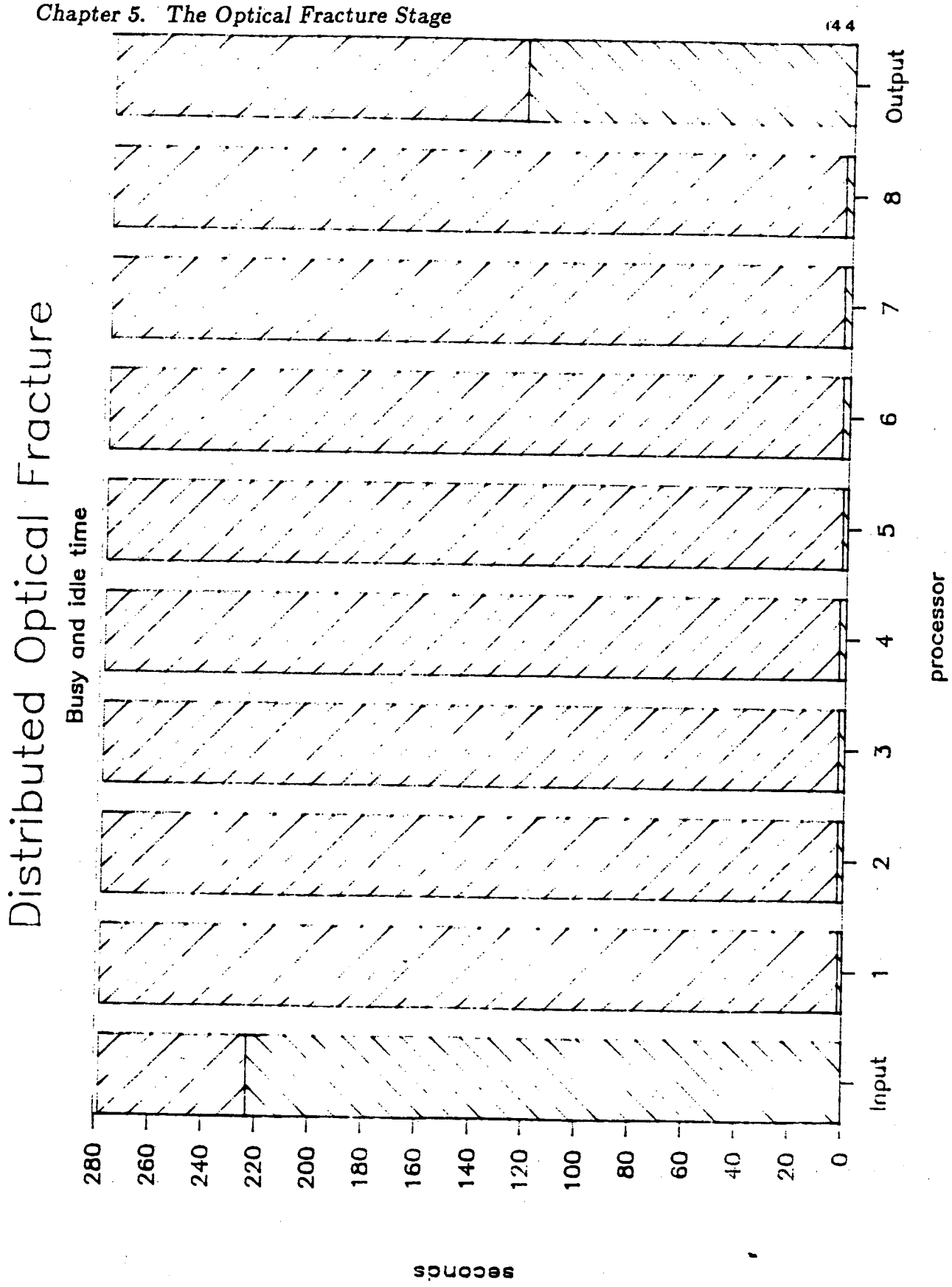
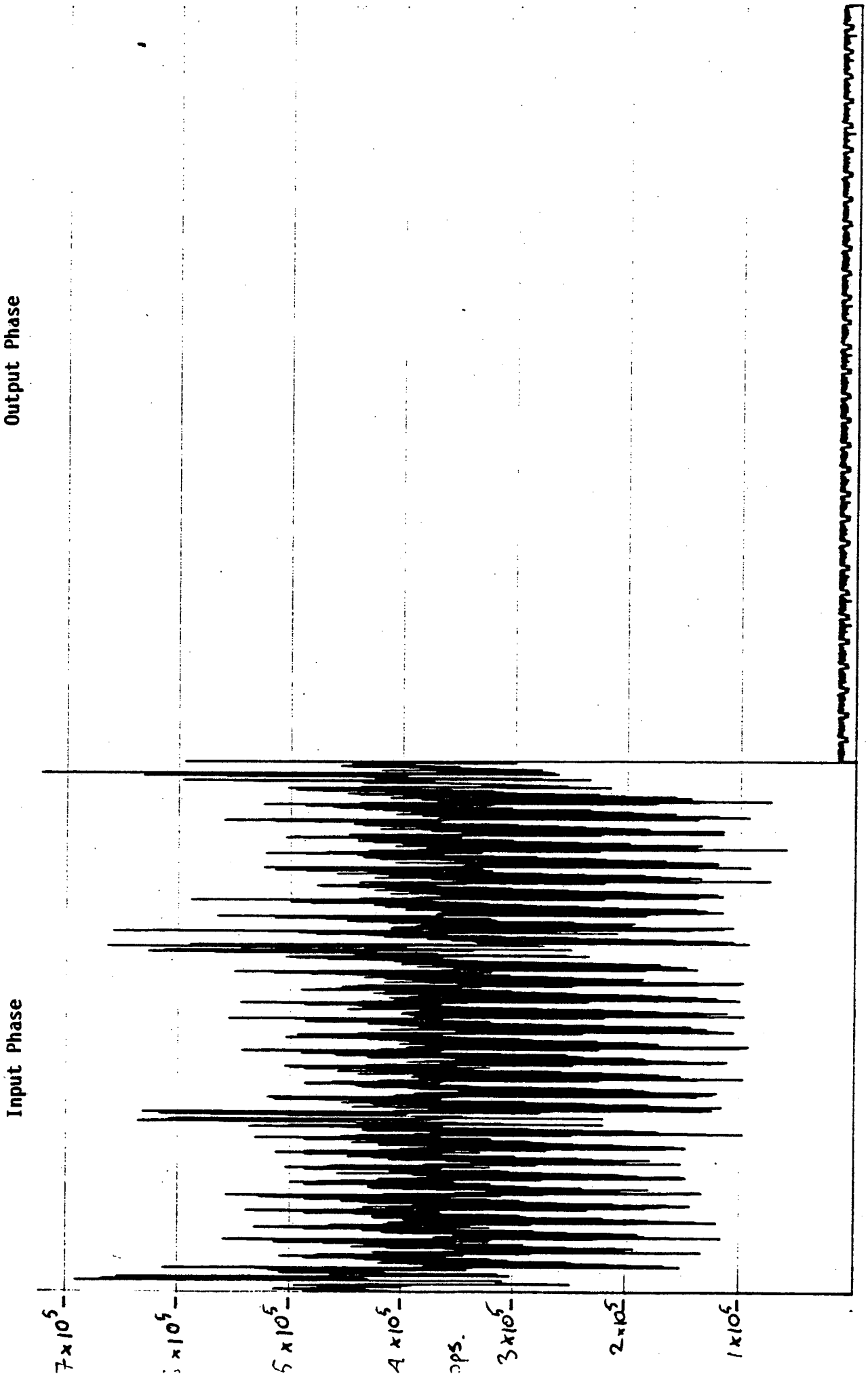


Table 5–6: Processor Loading


Output Phase
Input Phase
7x10^5
5x10^5
4x10^5
3x10^5
2x10^5
1x10^5

Chapter 6

Conclusions

This thesis presents a series of techniques which improve the absolute performance of the process of pattern generation. This has been achieved by addressing one particular level of implementation, coarse-grain loosely coupled concurrency. The performance improvements are totally independent of any other implementation detail. There is no reason why any implementation given here could not coexist with any other sort of performance improvement *including further parallelism.* The method given is *not dependent upon* the basic algorithm, although in some cases the program has had to be amended. This is because the method is based not upon the algorithms but upon the *form of the problem.* Thus if a system were implemented on a distinct sort of parallel machine, for instance an SIMD machine such as the DAP, there is no reason why multiple (multi) processors should not be coupled together as detailed in this thesis to achieve even more speed up.

The implementations demonstrated and the next generation improvements suggested will be more than capable of handling the new generation of very large scale integrated circuits.

6.1 Chasing the Bottleneck

Although very much experimental, a speed up of *at least* four times has been achieved in *every* case, with greater speed up being achieved for those stages which are particularily CPU intensive. As has been stressed throughout, the implementation has scope for massive further improvements. In no case has communication presented a serious bottleneck, although some level of buffering has been assumed in the communications to smooth out bursts.

These improvements should take the form of further development by removing more and more of the bottlenecks in the system. Ultimately with careful enough coding the final bottleneck will be the speed at which data may be transferred to and from magnetic media, with some added cost for the recombination of the data.

This means that the major drawback with this research (and presumably all research) was knowing when to stop. With hindsight it is always easy to know exactly how the development should have taken place and to know what improvements could be made to make the system even better. The motivation throughout this research was to *prove the validity* of the ideas developed to parallelise the task of VLSI pattern generation. The further improvements to the system, both in terms of making the code more robust, and in terms of the considerable extra speed-ups which are still achievable, are purely a matter of software engineering. The development of existing ideas and programs such as those developed in this thesis, although not easy, is a well understood part of software engineering.

6.2 Architectures

Unusually for research in parallelism, none of the resultant algorithms are dependent upon a specific target architecture, but will run on any architecture which obeys the (architecturally loose) design constraints. This is because the methods developed have been targetted against loose, general limitations not tight, architectural ones.

There can be no doubt that the era of parallel processors being available in quantity has arrived. Each new parallel processing system presents a new architecture and a new method of programming. In this thesis there has been no emphasis put on single *hardware* architectures, but there has been, as a recurring theme a system architecture consisting of dedicated input and output processors and as many 'work' processors as can be supported in one-to-many and many-to-one communication with them. Any hardware which supports this configuration would be suitable.

In particular the most interesting of the possible architectures is the one which is well suited to silicon design centres. It is very common for design centres to have many workstations interconnected with a local area network and a central main-frame, very often used as a simulation engine and for keeping overall control of developing designs. The envisaged architecture is for the main-frame to handle the function of the input and output processors, and the workstations to handle the processing. See Figure 6–1.

Very often workstations are underloaded and the processing could run as a low priority background job. The importance of such an arrangement is that no further processing power is required above that which exists already, a large proportion of the load imposed by pattern generation being removed from the (usually overloaded) main-frame and placed upon the (usually underloaded) workstations.

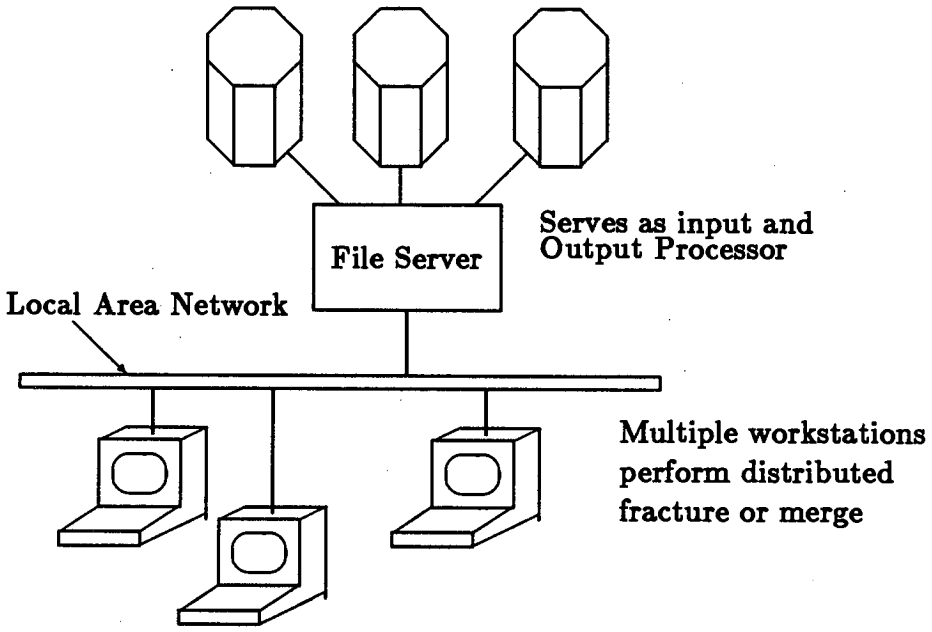


Figure 6–1: Workstation based Architecture

6.3 Further Work

As noted above the next stage of development of the system is the removal of the current crop of bottlenecks which absorb so much of the sequential processing power that they reduce the ultimate amount of useable parallelism. For the merge stage this is the normalisation and area calculation stage, for the fracture systems the recombinations of the output into the correct order. Appendix two details an implementation based upon such a system, incorporating many of the improvements suggested in Chapter 3.

In order that future implementations take advantage of the presented systems' capability to function on many different parallel architectures, it would be sensible to implement all the parallelism in terms of one of the public domain remote procedure call implementations which are currently becoming more and more numerous.

Obviously pattern generation is not the only time consuming part of the process of integrated circuit design which is amenable to performance improvement by the use of parallelism. It is hoped that some of those processes which are similar in many ways to pattern generation (in particular design and electrical rule-checkers) are amenable to parallel implementation by use of similar techniques to those outlined in this thesis.

There is a very obvious alternative (and indeed supplementary) method of parallelising pattern generation, which is to perform it on every layer simultaneously. Thus if a design was being processed to generate seven masks, seven separate systems could be used to perform the processing time in the time taken to process the slowest layer. The amount of data on layers tends to vary considerably and so the resulting speedup would not as much as might be anticipated. Of course there is no reason why such a technique could not be combined with the techniques developed in this thesis. In particular the parallel merge stage works best for large designs and so it might be possible to merge several layers concurrently on one multiprocessor - a small layer would be processed on one processing unit only, a medium sized layer on two or three and a massive layer on the remaining processing units. In this manner the best possible utilisation of machine resources and thus turnaround can be achieved.

Data processing for pattern generation is going to become more and more important as designs become larger and larger. The work described in this thesis should alleviate any speed-related problems in the immediate future but there are still vast areas to be invesitgated. These include finding better heuristics for the computationally intractable problems, getting better absolute time performance for the tractable, and the continual search for better accuracy and more graceful handling of the perpetual rounding problems.

Newer generations of mask patterning machines, and indeed all Pattern Generation equipment, will present new and greater challenges.

Appendix A

An outline of changes made to the Merge code

This appendix describes the amendments which were made to the Kilgour's polygon package [Kil86] to enable it to run (in emulation) as a parallel system.

A.1 The Original Code

The basic algorithm is scan line with the scan line moving *top down*. This can cause some confusion in that the view of the Y coordinate can be strange (bounding boxes for instance are kept top-left, bottom-right). The Algorithm has five stages.

1. The input geometry is parsed. Originally this was CIF [HS80], however many other formats, both artwork and PG, may be read. Which language is parsed is unimportant since, geometrically speaking, all the languages are similar. Since the input geometry may be hierarchical it is parsed into a hierarchical data-structure which is then flattened and the basic polygons handed to the second stage.

2. The geometry is normalised, coincident points are removed, the vertex order is reversed where needed and holes are removed and treated as

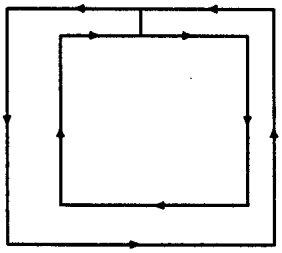


Figure A–1: Self Touching Polygons

separate sheets. CIF enforces no order on the vertices of a polygon, nor does it have any concept of holes. These have to be achieved by having 'self-touching-polygons' [NS80]. See Figure A–1.

The CIF polygons (and wires and flashes which have been converted into polygons during stage one) are converted to a normal form, for the polygon package this is as follows: A ***polygon*** in the polygon package consists of a doubly linked list of ***sheets*** (in arbitrary order) each of which is either a ***box*** (a Manhattan rectangle, this being used purely as a space saving device, one which is essential when remerging optically fractured input) or a doubly linked list of vertices. The *order* of the vertices determines whether the sheet has ***positive*** or ***negative*** sense - respectively is a sheet or a ***hole*** (note the ambiguous overload of the term sheet).

A sheet has vertices in a ***anti-clockwise*** order, a hole in a ***clockwise*** order. Thus the enclosed area is always to the ***left*** of the line between consecutive vertices.

Another function of the normalisation is to reduce all self-intersecting polygons to non-intersection.

3. A pre-processing stage is performed. This consists of finding all the local maxima of all the sheets and holes. A local maximum is where:

```
Sheet_Next_Y <= Sheet_Y %and Sheet_Last_Y < Sheet_Y
```

All these points are sorted and placed into the *Event List*, the sorting being done by Y and then X. The event list is a priority queue, a dynamic structure containing an ordered list of events. An *event* is where the scan line will stop and processing will occur. The most important events are at the start of an edge (produced by local maxima), the end of an edge, and the intersection of two edges. At any point the event list will not contain all the events, for instance at the start it will only contain start edge events. Processing of an event may cause generation of more events, these events will always be further down the event list (lesser Y or equal Y and greater X). Thus the event at the head of the priority queue will always be the event which should be processed next.

4. The scan line processing can now proceed. In addition to the event list, one other important structure is associated with the movement of the scan line down the image. The active edge list, a fully dynamic structure, is a doubly linked list (initially empty) of all the edges which cross the scan line at the time of the current event. There are performance gains to be had from using AVL trees or b-trees, but these are offset by the ease of removal of a series of edges from the list.

   Each edge in the active edge list is has associated with it a path (a list of vertices which form an incomplete sheet). Obviously each path is associated with only two edges.

   Assigned to each path (and thus every output sheet) is a wrap number. This may be computed by moving along the edge list incrementing a count, initially at zero, every time a positive edge (going top to bottom) is crossed and decrementing it for event negative edge. The union of the polygons is all the sheets with wrap number 1, the intersection all those with wrap number 2.

   Processing an event consists of updating the active edge and path lists appropriately, checking for intersections where necessary and if needed outputting a sheet, any further events generated are inserted in the event

list. Special care needs to be take in the case of several edges crossing at one point. If further processing is required (see below) outputting a sheet consists of appending it to an output polygon, otherwise the sheet is passed to stage six. The processing completes when the event list is empty.

5. If sizing is to be applied then it occurs at this stage. If bloating is applied then the acute angles may be clipped, See Figure A–2. After the sizing stages three and four are repeated

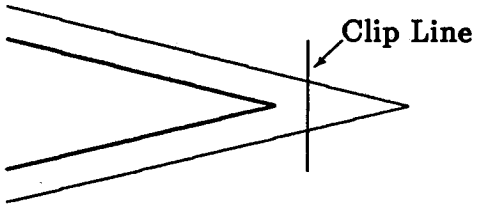


Figure A–2: Clipping Acute Angles

6. The output form is a dense, binary representation of holes and sheets. This Merged Pattern Data (MPD) may be viewed or plotted, passed to later pattern generation stages (fracture) or converted to any of the input formats. Some of these formats define a maximum edge count for polygons, furthermore large polygons can affect the performance of the optical fracture algorithm dramatically. Thus at output the sheets may be decomposed into smaller units.

The presence of stages one and six are noted only for completeness since they have no impact on the research presented here. The method in the decomposition used in stage six in particular may be very complex. The development of the decomposition algorithm was carried out in paral-

lel with this research and was incorporated after a frozen version of the Polygon Package was taken for development.

A.2 New Modules

Although alteration had to be made throughout the body of the polygon package only two new modules of any import had to be written. Since the original package was written in IMP [Rob86] so were the additional modules. Thus any code fragments have been given in psuedo-IMP.

The two modules were SPLIT, which (per processor) clipped the input shapes and STITCH which recombined region-spanning shapes on output. The main control module was substantially alterered and a minor module (TIMER) written which took care of simulated time.

Finally, as part of the work carried out on Ebeam fracture a new output module was developed which presented the same interface as the standard module. Thus the linker could be used to chose the output format. The standard output module was used for all timings. An overview of the new output format and justification for it is contained in Chapter 4. Although the standard output module worked in the simulation it would not do so were 'real' concurrancy involved thus a description of a different output module - not disimilar to the one actually written, is given.

A.2.1 The SPLIT Module

The input communication model used is that all processors receive continuous stream of shapes which cover the complete image area. Each processor also knows which the region of the image it covers. At this first stage the regions were allocated by trivially dividing the image in X and Y into equal sized rectangles. For every input shape each processor can quickly decide to:

- Discard the shape

- Use the shape

- Clip the shape

This is a fast operation since each shape has an associated bounding box which can be quickly compared with the region.

```
%if Shape_BB_Min_Y < Region_Max_Y %or
    Shape_BB_Min_X > Region_Max_X %or
    Shape_BB_Max_Y > Region_Min_X %or
    Shape_BB_Max_X < Region_Min_X %start
  { Ignore shape }
  %continue
%else %if Shape_BB_Min_Y <= Region_Min_Y %and
    Shape_BB_Max_Y >= Region_Max_Y %and
    Shape_BB_Min_X >= Region_Min_X %and
    Shape_BB_Max_X <= Region_Max_X %then
  Append Shape To Current List
%else
  Clip Shape
%finish
```

It is the clipping which forms the major part of this module. During the development of the SPLIT module, the emphasis was placed upon achieving an implementation which would be easy to design, develop and debug. The impact of the efficiency or otherwise of this system was perceived to be minimal since the clipping would take place in the merge processors, and be applied to a small subset of the input data.

The input to the clipping algorithm was a well formed sheet. In contrast to [SH74], the polygon was clipped to all four sides of the region in one pass. Each vertex of the input sheet was examined. If it lay within the clip region then it was added to the embryonic clipped sheet. If the vertex was outwith the clip region then it was projected onto the edge of the region if the previous or next vertex was within the region, otherwise it was ignored.

When a sheet reentered the clip region, great care was taken to distinguish separate sheets which are found as a result of the clipping, see Figure A–3. As

a result, the clipped sheets were well formed and could be passed directly to the merge system.

In the system presented in Appendix B, the clipping algorithm adopted is that described in [LB83].

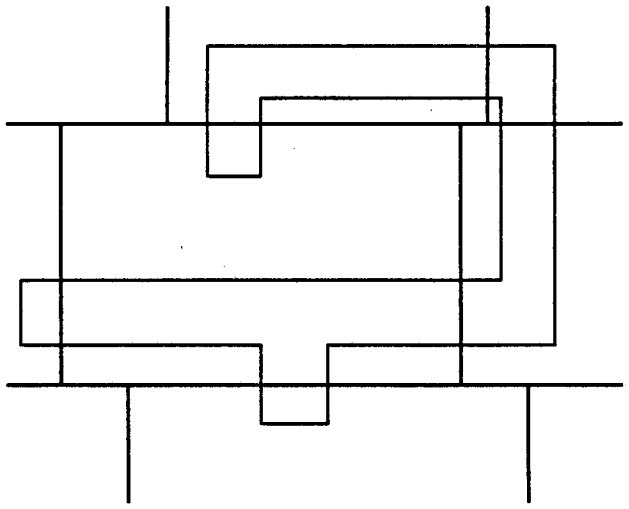


Figure A–3: Nasty cases when clipping

When a shape is output, it is sorted according to whether it is completely within the region, or impinges upon one (or more) of the boundaries. In the former case it is handed to the output module but in the latter it is handed to the STITCH module.

A.2.2 The STITCH Module

During the development of the SPLIT module (and in the system presented in Appendix B) this was purely a call on the merge routine. This had the obvious advantage of being an immediately usable module but it also worked remarkably well. It is unnecessarily complex and slow when the results are large, complex polygons which cover most of the design, so the STITCH module was written.

For each region four linked lists were kept, one each for the East, North, West and South edges. These were initially empty. Whenever an output shape touched one of these edges an entry was made in the appropriate list, which was kept sorted by increasing X for the North and South lists and increasing Y for the East and West. Associated with each entry was the corresponding vertex and a field which indicated whether the edge was outgoing or incoming. Stitching up a design took the form of scanning the relevant lists, combining the polygons on both sides as necessary. Special care needs to be taken at the start and end of the list. If the first element is of type 'outgoing edge', then it may be necessary to introduce a hole (see Figure A–4). If the final element in the list is in the corner, then the sheet may cover the region, and some extraneous points at the corner will need to be removed.

In order to minimise these difficulties, the East/West edges were combined first, leaving short fat slabs to be combined. The North/South lists for these slabs were joined together (so that they now spanned the complete design). Having done this the case of hole generation was reduced to checking whether the two vertices (North and South) forming an outgoing edge pair belonged to the same polygon (ie the polygons had been previously joined). Checking for covered corners took on the form of checking for (and removing) coincident points in the joined list.

Recent investigation has shown that STITCHing is not as important as may be immediately be assumed. In particular, as described in Chapter 4, not performing stitching presents a natural method of limiting vertex size, which can have an enormous impact on the functioning of later PG stages. The only places where STITCHing may be required are where slivers which might present difficulties to the mask maker are present at the edges of the regions, and where non-Manhattan geometry causes acute angles to be introduced when the masks are to be made by Optical equipment.

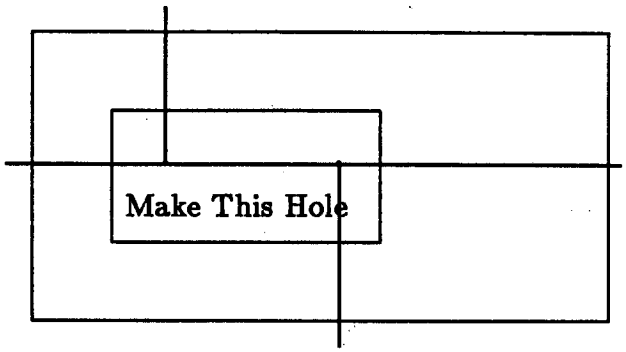


Figure A–4: Nasty cases when Stitching

A.2.3 The OUTPUT Module

The emulator works by running one processor to completion before starting the next. Thus by using the standard IO module an ordered output could be achieved.

In reality, of course, output sheets will be produced by all the processors concurrently and so the standard output module would produce output in an arbritrary order. As shown in Chapter 4 it can often be useful to keep the output from each merge processor separate. An immediate solution and the one adopted in Appendix B, is to have an output file per processor and to combine these files at a later stage. This is somewhat inelegant and can be inefficient. A better solution is to have one output file which is logically separated into several output streams.

This is very similar to the EFF files introduced in Chapter 4 and indeed is implemented on top of the same block based IO system. As for the EFF files (and similar to the 'backwards' data files used as the input to the parallel Ebeam fracture) the last datum in each block points to the first datum in the next block which contains data output by the same processor. The first block contains a series of pointers to the first block of data of each logical file. See Figure A–5.

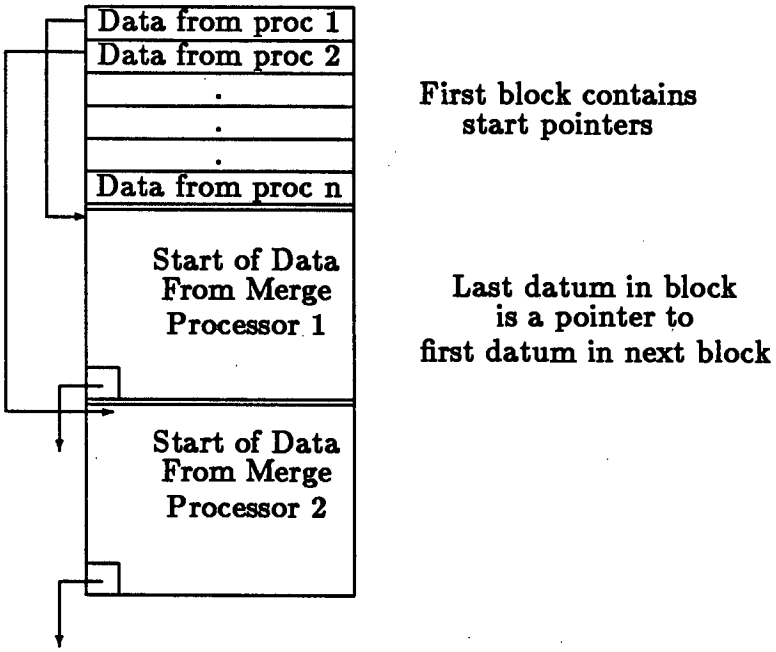


Figure A–5: Ordered Concurrent Output

Appendix B

Take Three. Parallel Merge on Real Hardware

The results given in this thesis are all based on emulations of a 'perfect', although loosely constrained system. This was the best way of doing the development since no hardware which remotely corresponded to the target was available at the start of the research. Furthermore, working one level removed from any actual hardware enforced concentration on the problem rather than on one particular hardware implementation.

Towards the end of the period of research two parallel systems which corresponded to the target architecture became available. At the same time it was felt that the research would benefit from having some real hardware to support the results demonstrated in the body if the thesis. Finally there was an overwhelming urge to put right some of the wrong design and implementation decisions which had resulted in less than the expected speed up.

The chosen stage to implement on real concurrent hardware was the merge stage, the development of which is detailed in Chapter 3. As well as being the most time critical of the applications some of the required programming alterations had been done, as described below, which made the parallel implementation much easier. As described in the final section, the two parallel systems used were a Vax/VMS Local Area Vax Cluster and a cluster of diskless SUN workstations.

B.1 Implementation

B.1.1 Background

During the development of emulated of the parallel merge system which is described in Chapter 3, it became apparent that the design partitioning, originally conceived as a method of parallelising the merge operation could be exceptionally useful in allowing the control of the memory requirements of the sequential merge system. To this end a development of the partitioning was adopted. This took advantage of the mistakes which had been made when developing the emulation and was done in such a way that subsequent parallelisation could be done easily.

B.1.2 Division and Recombination

The programming described in this section was performed by Kenneth Ferguson, working at Lattice Logic, under the general guidance of the author.

The basic merge program was amended in two ways. Firstly the design had to be partitioned into small segments such that the memory requirements would not be large, and secondly the data which crosses a segment boundary needed to be recombined.

Rather than divide the design in two dimensions as described in Chapter 3, only a one dimensional division was implemented. This had the advantage that only one pass was required to derive the partition boundaries. It was quickly established that the memory requirements for long, thin strips was much less than for short, fat strips. This can be seen intuitively since there are two data-structures associated with the X direction (the currently open paths and the active edge list) and only one in the Y direction (the event list). The size of the strips are established by equal vertex count, since this gives very good control of the memory requirements. The ability to have fixed width segments was also

included to allow segments to have the same width as the Ebeam segments, thus allowing the requirement for stitching to be relaxed.

The width of each segment is determined thus: The design is divided up into many *thin strips* and the vertex count within each one is determined during one pass over the data. These thin strips are grouped together until the sum of their vertex counts was equal to or greater than threshold value, which can be specified as a command line option, at which point the width of the first segment has been established. This process is iterated until the complete design has been covered with segments.

Each segment is then processed individually, with the processing moving from the left to the right. See Figure B–1. At this stage the input data still exists in a hierarchical format in the internal CIFSYS data-structure. This data-structure can be unwound into the flattened state many times, communication being via routine calls to deal with the geometrical primitives. One of the attributes of the data-structure is that prior to each unwinding a *clip window* can be set. When a cell instance is being expanded, if it is completely outside the current clip window the cell is not expanded. Thus the amount of data handed to the client program (which of course still has to apply clipping) is considerably reduced.

This facility was originally intended for a graphics viewing system where it is convenient to zoom into individual parts of a design as fast as possible, but was also used to great effect in the processing of the segments. The clip window is set to the current segment window prior to unwinding. As the data is passed to the merge system it is completely clipped to the segment window. As described in Chapter 3, the segment window is increased if sizing is being applied and the data clipped again after the sizing operation.

During output the shapes are classified as to whether they impinge upon the left of the segment window, the right of the segment window or are totally within the segment window. In the last case the shape is output immediately, otherwise the shape is added to the list of left seam and right seam shapes.

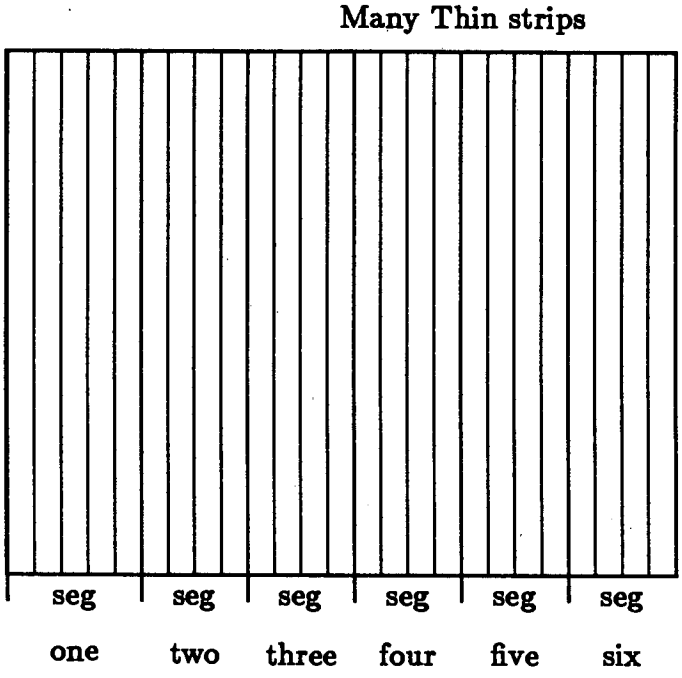


Figure B–1: Using thin strips to determine segment width

When the processing of one segment has been completed the left seam is re-merged, thus joining up abutting shapes. Extraneous points along the boundary are removed as described in Section 3.3.1. The left seam is then cleared, and the right seam shapes are moved in to the left seam to form the left hand side of any subsequent recombination operation. Thus accurate recombination is achieved.

B.1.3 The Parallel Implementation

The basic system can be broken up thus:

- Parse the input geometry
- Calculate the segment windows as described above
- Performing the merge and bias and (where needed) recombination on each stripe, whilst writing the output data to a temporary file.

- When processing has been completed, the temporary files are copied to the permanent file, a suitable header containing information such as bounding box and layer information being inserted.

In the parallel implementation each processor parses the input geometry. The segment windows are calculated thus: Each processor is assigned, for the purposes of segment window calculation a geometric partition. Thus if there are three processors and 3000 thin strips, the first processor establishes the vertex count within the first 1000 thin strips, the second the next 1000 and so forth. Obviously the clipping window can be applied to CIFSYS during this operation, resulting in a possibly faster segment area calculation. The processors then synchronise, and exchange the values for the thin strip counts. The segment window recalculation then proceeds as for the sequential system. Each processor therefore knows how many segments the design is going to be partitioned into and thus which segments it has to process. Obviously the number of partitions, and thus the amount of feasible concurrency, is controlled by the vertex per segment threshold. Recombination within the segments associated to one processor is performed as for the sequential system, but any shapes which impinge on the extreme boundaries of the any processor's window are written to a temporary file, separate from the main temporary file which contains the majority of the data.

When the processors have finished their allocated segments they synchronise and exchange the bounding boxes of their output data and the names of their temporary data files. One processor alone is responsible for the recombination of the boundary shapes, whilst another is responsible for concatenation of the data files and the adding of the bounding box information header. When the recombination is finished the processors synchronise again and the recombined data is finally appended onto the output data file. Figure B–2 gives a schematic of the system. Time proceeds downwards.

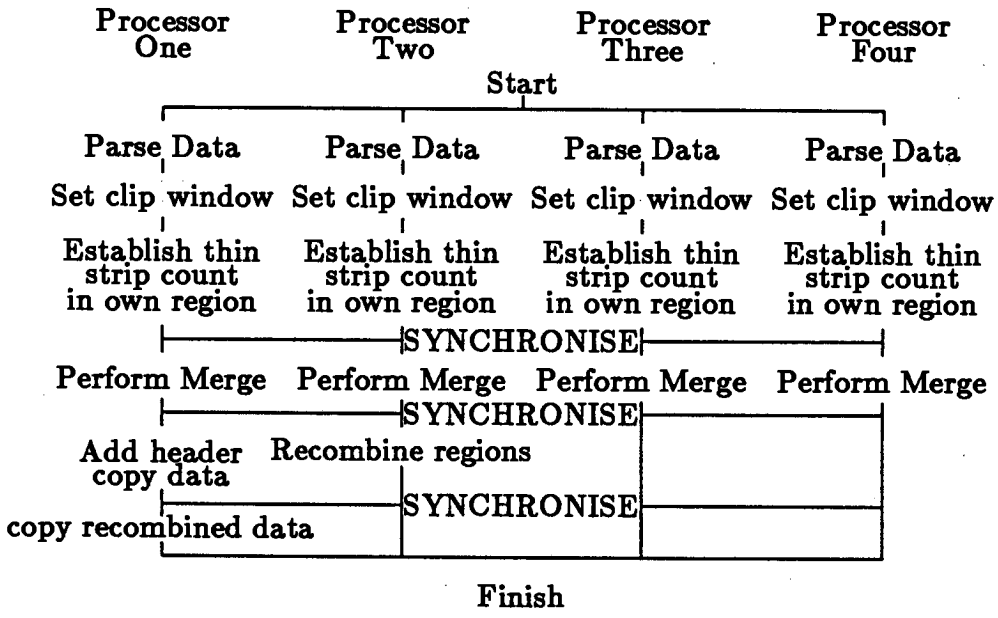


Figure B–2: Schematic of Parallel Merge

Communication

Although at a low volume, interprocessor communication forms an important part in this system - vertex counts, data windows and such like have to be freely available to all processors. Thus some sort of communication has to be available. Ideally this communication should not be targetted to one particular operating system. The method chosen was to use standard input and output to perform the communication, using a shared file server as medium. Furthermore it was assumed that the input and output data will reside on this shared filing system. Thus as well as the clusters described here, any distributed processing system with a common file server could be used as a suitable basis for parallel merge. This system has the advantage of being simple, highly portable and flexible. It lacks some of the speed that a special purpose communication package might have, but the amounts of data being handled are small and the speed requirements are not high, especially since in the system presented above communication is invariably associated with synchronisation. Obviously this

system requires that the file server can adequately handle multiple reads on data.

Synchronisation

The synchronisation was based on a simple flag system, whereby each processor raised a flag when it was awaiting synchronisation. When the flags for all the processors were raised, synchronisation had been achieved and processing could continue. As for communication the medium used was the shared file server. Each flag consisted of the existence of a file (with a unique name). Waiting for synchronisation became waiting for the existence of a series of files. Thus the general synchronisation routine became:

```
%routine Synchronise Processors

   %own %integer Number = 0    { How many times have }
   %integer i                  { we synchronised     }
   %string (63) Target

   Target = I to S(number,0)."Sync"
   Open Output (1, Target. I to S(My Processor Number,0))
   Select Output(1)
   Printstring("Nibble a Happy Warthog!")
   Newline
   ! in case the Operating system optimises
   ! away any empty files
   Close Output

   %for i = 1, 1, Number of Processors %cycle
      %while %not Exists(Target.I to S(i, 0)) %cycle
         Wait (5)  { seconds  - avoid busy loops }
      %repeat
   %repeat

   Number = Number + 1
%end
```

Again this has the advantages of flexibility, simplicity, portability and very simple debugging - if the system deadlocks it is easy to see, by the state of the

synchronisation files, the state of each processor. The requirement for synchronisation is not great - there are only three synchronisations in the total system and so its inefficiency is not important.

It should be stressed that the files used to provide the synchronisation should not be used for data transfer. Processing will proceed as soon as the synchronisation file *exists* by which time it may not have valid data in it.

B.2 Results

A major advantage of the system described above is that it has no machine dependent part and such can be swiftly ported to a range of machines so long as they support multiple access to a common file server. The following gives the results of porting the system described above to two particular systems.

B.2.1 Vax Cluster

The hardware here was a small Local Area Vax Cluster (LAVC) where the communication is achieved through a 10Mhz ethernet. The cluster consists of a VAX 11/780 with 8 Mb of memory and a micro Vax (μVax) with 13 Mb. This meant that the processing power was relatively well balanced. Both machines paged off disks local to themselves. The communication, synchronisation and data files were mounted on a disk attached to the μVax. This had a slight impact on the speed of access to the data by the 780. The cluster presented the user with machine specific, low priority, batch queues and so the the merge was performed by submitting one job into each of these queues. All processing was performed during the evening, when the effects of other users' load could be minimised (The 780 supported a massive daytime load, in contrast to the μVax and so the μVax would have had to wait for considerable periods).

Table B–1 gives details of the results. Two runs were performed one with a vertex per segment limit of 80,000 and one at 120,000. For each run the CPU

and elapsed time for a sequential run on each of the processors is given and the elapsed time only for the parallel processing. No factored speed up is given since it is unclear what the yardstick should be for an unbalanced cluster. In each case the the timings were taken for a merge and shrink of one layer of a large design. No stitching was applied during these runs, this being the default mode of running for the partitioning merge program. As described in the chapter on Ebeam fracture lack of stitching is an advantage for parallel Ebeam fracture.

	VAX 11/780		MicroVAX		LAVC
Vertex Threshold	CPU Time (h:m)	Elapsed Time (h:m)	CPU Time (h:m)	Elapsed Time (h:m)	Elapsed Time (h:m)
80,000	5:03	5:54	4:43	4:52	3:35
120,000	5:06	6:06	4:49	4:59	3:11

Table B–1: Actual time taken to process a design on a LAVC

B.2.2 Diskless Sun Workstations

The system used here was that presented in Chapter 6 namely a collection of workstations attached, again through a local area network (a 10 Mhz Ethernet), to a fileserver. The actual workstations were Sun model 3/110, each with 4 Megabytes of real memory. They all have virtual memory, with pagefaults being handled by the same fileserver as was used as the communication medium. In addition the fileserver itself could be used as a processing unit. The cluster was of 4 machines. Six runs were performed, being on the same data as was used by the VAX-cluster presented above. For vertex thresholds of 80,000 and 120,000 three runs were made. Firstly the standard product was run both with a local disk and a remote disk and then on the cluster. As can be seen by reference to the results given in Table B–2, the presence of a local disk had little effect on timing, in fact since the machine with the local disk was performing other tasks its elapsed time is slower. This confirms that memory is being well controlled.

As always the numbers given are limited by the accuracy of the operating system. In addition some of the figure may be greater than expected due to the impact of other users; on an open system such as this it is difficult to ensure that the machines were not supporting other users; this is especially true for the local disk machines, which was a file server for many other machines.

	Sun local disks		Sun remote paging		Cluster
Vertex Threshold	Elapsed (h:m)	CPU (h:m)	Elapsed (h:m)	CPU (h:m)	
80,000	5:28	5:08	5:04	5:03	1:30
120,000	4:50	4:44	4:45	4:43	1:35 ‡

In the run marked ‡one of the processors was heavily loaded and this affected the time taken by approximately 15 minutes.

Table B–2: Actual time taken to process a design on a cluster of SUNS

B.2.3 Final Points

These results adequately support the original decision to do the development upon an emulation. The predicted level of speedup has been achieved and the cost of communication has been kept well under control. In addition the first implementation upon actual concurrent hardware has been made much simpler and it has been possible to introduce many improvements. However it is worth emphasising a few points which arise.

Firstly in both the emulations above the processing power was somewhat similar and so a trivial partition of the problem could be achieved. Were there a more marked disparity in the relative power of the machines in the cluster, then the calculation of the division would need to be more complex. In the case both of the LAVC and the Sun cluster the most powerful processor was given the extra task of handling the control of the file server. This being in marked

disparity with the 'perfect' design where the IO was handled by a dedicated processor. However as detailed in Chapter 3, the merge stage was less bound by the power of the IO processors. As demonstrated in section B.2.2, there is also an advantage in having the processor with local control of the file server to perform the file concatenation.

The main drawback with the system is that if one of the processors has an extra load applied (for instance an interactive load or as in the case of the SUN system by operating system oddities), then all the other processor in the group are going to spend a great deal of time waiting for this processor, and so it is important to ensure either that the processors all have (roughly) the same load, or that the processor with the greatest load has the least work to perform as part of the merge cluster.

Bibliography

[AAI86] T. Asano, T. Asano, and H. Imai. Partitioning a polygonal region into trapezoids. *JACM*, 33(2):290–312, April 1986.

[ABJN85] D. Ayala, P. Brunet, R. Juan, and I. Navazo. Object representation by means of non-minimal division quadtrees and octtrees. *ACM Trans Graphics*, 4(1):41–59, January 1985.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS SJCC*, 483–485, 1967.

[Bai77] H.S. Baird. Fast algorithms for LSI artwork analysis. In *Proceeding of the 14th Design Automation Conference*, pages 303–311, June 1977.

[BB80] E.E. Barton and I. Buchanan. The polygon package. *Computer Aided Design*, 12(1):3–11, January 1980. also EUCSD CSR 44-79.

[BDHM84] D Bitton, D.J. DeWitt, D.K. Hsiao, and J. Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(16):287–317, September 1984.

[Ber86] N. Bergmann. A polygon fracture algorithm for opto-mechanical pattern generation equipment. In *Microelectronics '86*, Adelaide, Australia, May 1986.

[BHH80] J.L. Bentley, D. Haken, and R.W. Hon. *Statistics on VLSI designs.* Technical Report CMU-CS-80-111, Carnegie-Mellon University, Department of Computer Science, Pittsbugh PA, Aprill 1980.

[BO79] J.L. Bentley and T.A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9), September 1979.

[BOW83] J.L. Bentley, T.A. Ottman, and Widmayer. *Advances in Computing Research*, pages 127–158. Volume 1, 1983.

[Bur77] W. Burton. Representation of many-sided polygons and polygonal lines for rapid processing. *CACM*, 20(3):166–170, March 1977.

[Cha75] T.H.P Chang. Proximity effect in electron beam lithography. *Journal of Vacuum Science and Technology*, 12(6), Nov/Dec 1975.

[Cra75] B.J. Crawford. Design rule detection for IC's using graphical operators. In *Proc. Second Annual Conference on Computer Graphics and Interactive Techniques*, pages 168–176, ACM SIGGRAPH, June 1975.

[Den86] P.J. Denning. Editorial - parallel computing and its evolution. *CACM*, 29(11):1163–1167, December 1986.

[DLS86] J.T. Deustch, T.D. Lovett, and M.L. Squires. Parallel computing for vlsi circuit simulation. *VLSI Design*, 46–52, July 1986.

[Fer87] K. Ferguson. Memory requirements for low VM polymerge. Personal Communication, March 1987.

[Fly72] M.J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, C(21):948–960, 1972.

[GG83] K.D. Gourley and D.M Green. A polygon to rectangle conversion algorithm. *IEEE computer Graphics*, 3(9):31–36, January/February 1983.

[Gut84] R.H. Guting. Dynamic C-oriented polygonal intersection searching. *Information and Control*, 63:143–163, 1984.

[Heg82] A. Hegedus. Algorithms for covering polygons by rectangles. *Computer-aided design*, 14(5):257–260, September 1982.

[Hen77] G.H Henriksen. Reticles by automatic pattern generation. *SPIE Semiconductor Lithography*, 2:86–95, 1977.

[HJ84] R.W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hiller, Bristol, 1981.

[HMS*86] J.P. Hayes, T. Mudge, Q.F. Stout, S. Colley, and J. Palmer. A microprocessor-based hypercube supercomputer. *IEEE micro*, 6–17, October 1986.

[HS80] R.W Hon and C.H Sequin. *A Guide to LSI Implementation.* Technical Report SSL-79-7, XEROX - Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, January 1980.

[J82] Johnson D.S. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 3:182–195, 1982.

[Kil82] A.C. Kilgour. *A Span-Line Approach to the Polygon Intersection Problem.* Technical Report CSC/82/R6, Glasgow University Computer Science Department, December 1982.

[Kil85] A.C. Kilgour. *Parallel Architectures for High Performance Graphics Systems.* Technical Report CSC/85/R5, Glasgow University Computer Science Department, 1985.

[Kil86] A.C. Kilgour. *Polygon Processing for VLSI pattern generation.* Technical Report, Glasgow University Computer Science Department, 1986.

[KLS86] N.P. Kronenberg, H.M. Levy, and W.D. Strecker. VaxClusters: a closely-coupled distributed system. *ACM Trans. Comput.*, 4(2):130–145, May 1986.

[Lau78] U. Lauther. 4-Dimensional Binary Search Trees as a means to speed up associative searches in design rule verification of integrated circuits. *Journal of Design Automation and Fault Tolerant Computing*, 2:241–247, 1978.

[LP84] D.T. Lee. and F.P Preparata. Computational geometry - a survey. *IEEE Transactions on Computers*, C-33(12):1072–1101, December 1984.

[MC80] C.A. Mead and L. Conway. *Introduction to VLSI Design. Addison-Wesley Series in Computer Science*, Addison-Wesley, October 1980.

[LB83] Liang Y. and Barsky B.A. An Analysis and Algorithm for Polygon Clipping *CACM.*, 26(11):868–887, November 1983.

[NP82] J. Nievergelt and F.P. Preparata. Plane sweep algorithms for intersecting geometric figures. *CACM*, 25(10):739–747, October 1982.

[NS80] M.E. Newell and C.H. Sequin. The inside story on self-intersecting polygons. *VLSI design (was Lambda)*, 20–24, 2nd Quarter 1980.

[OW86] T. Ottman and D. Wood. Space economical plane sweep algorithms. *Computer Vision, Graphics and Image Processing*, 34(1):35–51, Aprill 1986.

[Rob86] P.S. Robertson. *The Imp Language, A reference Manual.* Lattice Logic, Edinburgh, Scotland, 1986.

[Sei82] L. Seiler. A hardware assited design rule checker. In *19th Design Automation Conference*, pages 232–235, 1982.

[Sei85] C.L. Seitz. The cosmic cube. *CACM.*, 28(1):22–33, January 1985.

[Sha78] M.I. Shamos. *Computational Geometry.* PhD thesis, Yale University, May 1978.

[SH74] I.E. Sutherland and G.W. Hodgman Reentrant Polygon Clipping *CACM.*, 17(1):32–42, January 1974.

[Sho70] W. Shooman. Orthogonal Computing. In *Parallel Processor Systems, Technologies and Applications*, chapter 15, pages 297–311, Spartan Books, New York, 1970.

[Sho73] J. E. Shore. Second thoughts on parallel processing. *Computers and Electrical Engineering*, 1:95–109, 1973.

[Slo67] D.L. Slotnik. Unconventional systems. In *AFIPS Conference Proceedings*, pages 477–481, 1967. Number 30.

[SR87] J.A. Shoeffel and M.L. Rieger. Economics of fast turnaround wafer production. *VLSI design*, VIII(2), February 1987.

[TBH82] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architecture. *Computing Surveys*, 14(1):93–143, March 1982.

[Tra85] J. Traub. Photomasks - Optical or Ebeam. *Silicon Design*, 2(9):5, September 1985.

[Wei80] K. Weiler. Polygon comparison using a graph representation. *A.C.M. Computer Graphics*, 3(19):10–18, July 1980.

[Whi81] T.E. Whitney. *A Hierarchical Design Rule Checker.* Technical Report 4320, Caltech (CS) Silicon structures project, May 1981.

[Whi85] T.E. Whitney. *Hierarchical Composition of VLSI Circuits.* PhD thesis, California Institute of Technology, Pasadena, California, May 1985.

[Wil80] J.A. Wilmore. A hierarchical bit-map format for the representation of IC masks. In *17th Design Automation Conference*, pages 585–590, 1980.