Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation

Murray I. Cole

Ph. D.

University of Edinburgh

1988
Abstract

In the past, most significant improvements in computer performance have been achieved as a result of advances in simple device technology. The introduction of large scale parallelism at the inter-processor level now represents a viable alternative. However, this method also introduces new difficulties, most notably the conceptual barrier encountered by the user of such a system in coordinating many concurrent activities towards a single goal. Thus, the design and implementation of software systems which can ease this burden is of increasing importance. Such a system must find a good balance between the simplicity of the interface presented and the efficiency with which it can be implemented. This thesis considers existing work in the area and proposes a new approach.

The new system presents the user with a selection of independent “algorithmic skeletons”, each of which describes the structure of a particular style of algorithm. The user must describe a solution to a problem as an instance of the appropriate skeleton. The implementation task is simplified by the fact that each skeleton may be considered independently, in contrast to the monolithic programming interfaces of existing systems at a similar level of abstraction.

The four skeletons presented here are based on the notions of “recursive divide and conquer”, “task queues”, “iterative combination” and “clustering”. Each skeleton is introduced in terms of the abstraction it presents to the user. Implementation on a square grid of autonomous processor-memory pairs is considered. Finally, examples of problems which could be solved in terms of the skeleton are presented.

In conclusion, the strengths and weaknesses of the “skeletal” approach are assessed in the context of the existing alternatives.
Acknowledgements

Many thanks are due to Gordon Brebner, who supervised this work. He provided encouragement and advice at the appropriate moments, carefully read and criticised numerous documents and was always prepared to listen to my ideas. Most importantly, Gordon has that invaluable asset of the good supervisor, an ever open door. I am also grateful to my father for proof-reading the first draft.

Finally, my thanks go to the Science and Engineering Research Council who funded my years as a post-graduate student and to the University of Glasgow with whose facilities the final revisions to this thesis were made.

Declaration

I declare that this thesis was composed by myself and that the work which it describes is my own.
# Table of Contents

1. Generating and Controlling Parallelism .................................. 1
   1.1 Introduction ........................................................................... 1
   1.2 The Spectrum of Existing Systems ........................................... 4
   1.3 Conclusions ........................................................................... 29

2. Algorithmic Skeletons – A New Approach .................................. 32
   2.1 The Abstract Machine ......................................................... 32
   2.2 Parallel Hardware ............................................................... 35
   2.3 The Implementation ............................................................ 41

3. The Recursive Divide & Conquer Skeleton ................................ 46
   3.1 Abstract Specification ......................................................... 46
   3.2 Implementing the Skeleton ................................................... 48
   3.3 Analysis of Implementations ............................................... 56
   3.4 Partial Trees ......................................................................... 63
   3.5 Examples .............................................................................. 66
Table of Contents

4. The Task Queue Skeleton
   4.1 The Abstract Specification .............................................. 72
   4.2 The Structure of an Implementation .................................... 77
   4.3 Implementing the Queue .................................................. 82
   4.4 Implementing the Data Structure ....................................... 98
   4.5 Summary ......................................................................... 106
   4.6 Examples ......................................................................... 107

5. The Iterative Combination Skeleton ....................................... 116
   5.1 Abstract Specification ..................................................... 116
   5.2 A Parallel Implementation Technique ................................... 119
   5.3 Implementation on an Idealised Grid ................................... 121
   5.4 Fixed Size Grids and Redistribution .................................... 131
   5.5 Redistribution with a Shortage of Objects ............................ 147
   5.6 Alternative Approaches .................................................... 153
   5.7 Examples ......................................................................... 155

6. The Cluster Skeleton .......................................................... 160
   6.1 An Alternative Approach to Skeleton Construction ................. 160
   6.2 Motivating a New Skeleton ................................................. 161
   6.3 Implementing the Skeleton ................................................ 167
   6.4 Examples & Discussion ..................................................... 176
# Table of Contents

7. Conclusions  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 The Case for Skeletons</td>
<td>182</td>
</tr>
<tr>
<td>7.2 Future Directions</td>
<td>185</td>
</tr>
</tbody>
</table>

Bibliography  

188
Chapter 1

Generating and Controlling Parallelism

1.1 Introduction

The task of designing and implementing any part of some computer system involves a process of abstraction. The facilities provided by an existing level are used to construct an implementation of an abstract level with its own, more desirable, properties. In a complete system, many such levels are involved, ranging say from the design of transistors using the physical characteristics of semi-conductors to the provision of highly specialised user interfaces built upon some underlying level of software.

Typically, an abstraction sacrifices a certain degree of freedom in return for a more useful and appropriate set of resources. In practice, this is reflected by a loss in performance of applications designed at higher levels over that which could be achieved (in theory) by direct implementation at a much lower level. Thus, a high level program subjected to a series of automatic compilations cannot be expected
to run as quickly as some hypothetical alternative solution, written directly in micro-code for the same machine and making optimum use of every available short cut. On the other hand, the original problem may be so complex as to make the latter course impossibly difficult. In this way, the process of abstraction can be seen to make a wider range of solutions accessible in practice (though not, of course, in theory).¹

The inevitable decline in performance associated with implementation has an important corollary, namely that the bounds upon the level at which it becomes impractical to build further abstractions are dictated by the absolute performance achievable at the hardware level. As hardware power and reliability increases, it becomes reasonable to move higher and higher levels of machine from the realms of theory into practice. For example, while it would have been possible to conceive of implementing an applicative language interpreter on typical 1950’s hardware, the resulting performance would have been uninspiring to say the least. Run essentially the same program on a machine hundreds or thousands of times faster and the abstraction suddenly provides a very useful, flexible tool.

The historical trends in hardware technology have been towards dramatically increased miniaturisation, speed and reliability. Complex components can now be mass produced at low cost. For the computer scientist, probably the most important development has been the erosion of the distinction between processor and memory technology, and the erstwhile mismatch in speed and cost between them. Computer architects may consider processing elements to be as readily available as were memory cells traditionally. Further technological developments will emphasise the new freedom. The revolutionary feature of VLSI (and what lies beyond) is not the increase in straightforward processing speed which it provides,

¹This is to say nothing of the associated gains in portability and clarity.
although this certainly has an important place in the evolution of traditional systems. Far more significant is the fact that it is now quite possible to build computers in which thousands of processors operate concurrently to solve a single problem. It is now practical to increase raw performance by replication as well as miniaturisation.

As an idea this is nothing new. In the very first issue of the Communications of the ACM, Saul Gorn [21] notes that:

"We know that the so-called parallel computers are somewhat faster than the serial ones, and that no matter how fast we make access and arithmetic serially, we can do correspondingly better in parallel. However access and arithmetic speeds seem to be approaching a definite limit ... Does this mean that digital speeds are reaching a limit, that digital computation of multi-variate partial differential systems must accept it, and that predicting tomorrow's weather must require more than one day? Not if we have truly-parallel machines, with say, a thousand instruction registers."

Thirty years on, the difference is that we can now construct such machines. However, Gorn also recognised the challenges that would be posed to the system designer by the new parallel computers:

"But visualise what it would be like to program for such a machine! If a thousand subroutines were in progress simultaneously, one must program the recognition that they have all finished, if one is to use their results together. The programmer must not only synchronize his subroutines, but schedule all his machine units, unless he is willing to have most of them sitting idle most of the time. Not only would programming techniques be vastly different from the serial ones in the
serial languages we now use, but they would be humanly impossible without machine intervention."

Where previous developments have slotted into the existing system hierarchy with ease, providing increased performance within the recognized framework, parallel hardware asks new questions. To what extent should the new found concurrency at the low level be reflected in the abstractions built on top? If the answer is "substantially" then how is parallelism to be presented? If "not at all" then is it possible to harness the new processing power to simulate existing structures, but faster? How much faster? Such questions underpin the area surrounding the work presented in this thesis.

We begin in this chapter by considering the motivation behind and approaches taken by a variety of existing projects in the field and their motivating principles. The lessons to be learned lead, in chapter 2, to the proposal of an alternative approach and a discussion of the model of parallel hardware chosen as a foundation. The remainder of the thesis presents a detailed investigation of one possible implementation of the new abstraction given the selected hardware.

1.2 The Spectrum of Existing Systems

In this section we investigate a variety of approaches which have been taken in addressing the problem of building higher system levels upon parallel hardware. In an attempt to enforce some order upon a widely divergent set of examples, the systems are considered in three loose categories.

The level of abstraction in the first category is high. Users of such machines are not required to deal with parallelism at all and need have no knowledge of the implementation to make use of the system. In the second category the
Chapter 1. Generating and Controlling Parallelism

degree of abstraction is reduced. Here, users are required to present explicitly parallel solutions. However, in doing so they are allowed certain freedoms not afforded directly by the hardware. The third category is something of a catch-all. Following the progression through the previous categories it contains systems in which the user is required to specify parallel solutions in a style very close to that present in the actual hardware. It also includes systems which might fall into either of the other two categories, were it not for the facts that their domains of applicability are very restrictive, and that their efficient use requires considerable understanding of both hardware and implementation.

As seems to be the way with any such artificial classification, practical examples have a habit of disregarding it. Thus we find projects which deal with machines apparently firmly rooted in one category adding ad hoc features which move it towards another. However, since the classification is only used as a starting point for discussion (and does not imply any qualitative judgement) this is not serious.

The main difficulties in implementing any parallel system arise from the requirements for problem partitioning and distribution (i.e. the identification of parallelism), data or code sharing, communication and synchronization. The systems presented in the three categories will be discussed with respect to these issues.

1.2.1 Highly Abstract Machines

In this section we discuss systems in which the abstract machine presented to the user is entirely devoid of any notion of parallelism and of the implementation mechanism (or almost so). They are based upon functional, logical and data-flow models of computation.
Chapter 1. Generating and Controlling Parallelism

Functional (or “applicative”) languages have been a focus of interest for some time (especially since Backus’ Turing Award lecture [7]). While there has always been an acknowledgement that these implicitly present much scope for parallel execution, it is only more recently that the practical problems of implementation have begun to be addressed. This is equally true of data-flow computing. Logic based languages and their implementation are relative newcomers. This discussion begins with an informal introduction to the foundations and properties of the various styles and their implementation at an abstract level. The main problems encountered in moving to a realistic implementation are considered and some aspects of contemporary approaches are evaluated. Most of the introductory material is taken from existing surveys [17,19,32,37,39,65,72].

All truly functional languages are based on the lambda calculus. This is a very simple, but powerful language of expressions and transformation rules on expressions. The only objects present are identifiers, single argument function definitions (“abstractions”) and applications of functions to arguments. A “program” consists of a collection of such objects and its execution amounts to applying a top-level function to some argument and evaluating the result. This involves function application, the only operation present, by which a function-argument pair is replaced by a copy of the function body (from its definition) with occurrences of the “dummy” or “free” variable replaced by copies of the actual argument (which may of course itself be a function application). Functions and arguments may be nested with parentheses. This simple system allows higher order functions (i.e. functions having other functions as arguments and results) to be defined and it can be shown that the lambda calculus has as much power as any other fundamental computing mechanism (e.g. Turing machines) in the sense of being able to compute any function normally held to be “computable”.

Just as imperative programming languages are clearly founded on the Turing machine model with its notions of state and store, so the more recognizable
functional languages are built from the simple lambda calculus. Features such as multiple argument functions, localised definitions, lists and operators upon them may all be defined as lambda expressions. For example, see [39] for a rather unusual implementation of integers.

Importantly, none of this abstraction removes any of the properties of the lambda calculus. They have functions as “first-class citizens”, which may be passed around just like any other object. In contrast to the concepts of store, state and control, there is only the notion of “objects” (e.g. function applications) which have values associated with them which are constant throughout the “lifetime” of the program. This property is known as “referential transparency” and contrasts with the variable objects of imperative programming which may have their values altered by assignment. Thus, from the outside, a functional program is a simple function definition which refers to other functions in its body. A “call” of the program involves supplying arguments to this function and “execution” consists of using the function definitions (effectively using the application by substitution technique from the lambda calculus) to obtain an alternative (but equivalent, by referential transparency) representation of the function and arguments pair. This “output” is simply a more useful representation of the original program and “input”. The important point here is that an implementation may progress from the initial to the final representation in any fashion which maintains the equivalence. In particular, it should be possible to execute parts of the transformation concurrently since the conventional problems associated with changes of state have been discarded along with the notions of state and store themselves (at least at this abstract level).

As has been noted, the semantics of the lambda calculus (and hence those of functional languages) allow some scope for varying the technique of evaluation (or transformation). The fundamental method is normal order evaluation. Starting with the original function and argument, the leftmost function application is
Chapter 1. Generating and Controlling Parallelism

considered. This is replaced by a copy of the function body in which references to the dummy argument are replaced with copies of the actual argument. The process is repeated with successive new leftmost applications. Eventually the representation expands until the only applications left are of primitive functions with no expansions (e.g. arithmetic operators). These are gradually replaced by representations of their results and the whole expression contracts until it represents some object suitable for output. In this type of scheme the computation is often said to be "data-driven" since operators are evaluated when their arguments are available, whether they will be needed or not. This also sometimes described as being an "innermost computation rule". The ubiquitous example is that both branches of a conditional statement will be evaluated even though one must eventually be discarded. It also means that multiple copies of the same argument will be evaluated (necessarily to the same result) independently, due to the use of substitution before evaluation. This can be thought of as a kind of call-by-value parameter passing.

The technique known as applicative order or eager-beaver evaluation avoids the multiple repeated evaluation problem by evaluating arguments before substitution. After argument evaluation (from the leftmost application) the reduced argument is substituted into the function body as before. The same element may be re-calculated at some other "remote" part of the program but multiple evaluations for the same substitution sweep have been eliminated. This method has been proved to give the same result as normal order evaluation with the exception of certain instances for which normal order evaluation terminates but applicative order does not. Typically these involve the evaluation of infinitely looping but unnecessary arguments. It is possible to envisage a parallel implementation with all locally leftmost arguments being evaluated in parallel.

A further extension known as "demand driven" evaluation maintains the efficiency of applicative order but inherits the termination properties of normal
order evaluation. Now, rather than evaluating the argument before substitution, pointers to the unevaluated argument are substituted into the function body. Evaluation of the argument is only performed if and when it is essential to allow overall evaluation to proceed. The argument is evaluated and subsequent references to other pointers can pick up the reduced value with no further effort. Since two processors evaluating and updating the same element via different pointers are guaranteed to produce the same result, the avoidance of such doubling up is not an issue in ensuring correctness. Applicative and demand-driven schemes are often said to have "outermost computation rules", with something akin to "call-by-name" or "call-by-reference" parameter passing.

A final step along the road leads to the logical conclusion of this sequence of methods, namely "lazy evaluation". Here, the method is also demand-driven. However, rather than evaluating arguments completely when they are required, only as as much of the argument as is immediately necessary is calculated. This allows the specification of infinite data structures (which are never completely evaluated) and the suspension of the evaluation of conditional branches until the condition itself has been evaluated. This could be very important when processing resources are limited in some way (as is inevitable in any real machine) and should not be wasted on speculative computation. Some schemes do away with pointers altogether, preferring to keep an "environment" containing pairs of (identifier, value) bindings. Here, application involves addition of an element to the environment, rather than the copying of a pointer to it.

The most important move in this sequence of evaluation methods is probably that from normal to applicative order. The crucial change is from application by direct substitution to application by reference. These two classes are mirrored in the range of proposed implementations. String oriented substitution machines perform application by direct copying of text, while graph reduction machines ex-
pand and reduce graphs of nodes representing functions and edges corresponding to application pointers.

A further addition to the graph reduction stable centres around the use of combinators [67]. In functional programming languages, bound variables are introduced to direct arguments into the bodies of function definitions. During execution, certain arguments are often superfluous. This leads to either unnecessary copying of pointers or correspondingly to the maintenance of never referenced pairs in the environment. The combinator approach transforms the bodies of all function definitions (before execution) into an equivalent special form without free variables (hence removing the associated problems). The revised definitions are constructed with the help of a small set of special functions known as “combinators” or “constant applicative forms”. These have the effect of steering arguments only to positions in the graph where they are actually required. The combinator version may be seen as a kind of compiled form of the original program. Associated with the combinators are rules which specify the manipulations to be performed on the graph in order to “evaluate” each combinator. Machines manipulating these combinator graphs are usually called combinator reduction machines and have many similarities to graph reduction machines in the ways in which graphs are expanded and contracted.

The data-flow approach arrives at a point in some ways similar to graph reduction from a different starting point. Here the underlying principle is the representation of a computation as graph of “operator” or “instruction” nodes connected by edges along which data items flow. Each node receives data “tokens” along its input edges, performs some simple calculation and distributes resultant data tokens on its output edges. The control mechanism is that a node may only perform its operation once it has received data tokens on all of its inputs. In a “static” system it must also wait for its previous outputs (if any) to be consumed, since at most one token is allowed to exist on an edge at any time.
Chapter 1. Generating and Controlling Parallelism

In "dynamic" systems this requirement is relaxed to allow queuing of tokens on edges. Thus nodes may operate in parallel, subject only to the availability of data. A typical data-flow graph will be re-entrant (cf. loops in imperative programs). In any realistic system there will typically be more operator nodes in the graph than available processors. The processes of associating output tokens with appropriate operator nodes and of deciding which are ready for execution is known as "matching". Ready operators must then be selected for actual execution. These processes are usually separated (at least in principle) from the "execution units".

The important difference between this approach and those discussed above is that whereas a graph reducer manipulates the graph by modifying both data and the "instruction code" itself (eg. combinators) a data flow graph is statically defined by the program and only data is manipulated. Typically, languages built upon these concepts are still "functional" in some respects (eg. no destructive assignment). They do not however inherit the ability to describe higher order functions and are often dressed up to resemble (at least superficially) sequential imperative languages [4]. The compilation process from this form to the underlying data-flow graph has some similarity to the process of expansion in normal order graph evaluation and it is not therefore surprising that innermost computation graph reduction methods are often described as being data-driven. The movement of data by pointer manipulation and substitution is comparable with the movement of data along edges in the data-flow graph. On the other hand there is no real parallel in data flow to the dynamic restructuring of the graph which takes place in demand driven graph reduction machines.

Logic languages (which effectively means Prolog at present) are based on Horn clauses, a restriction of first order (predicate) logic. The model of computation centres on the definition and investigation of relationships described as predicates, among data objects described as arguments to these predicates. The truth of a predicate, given some arguments, is specified as the disjunction of a collection of
"clauses". Each clause is specified as the conjunction of a collection of predicates, each over the original arguments or some subset of them. Additionally, certain simple predicates will be deemed to be true for certain specific arguments. These provide a basis upon which more sophisticated deductions may be built.

In similar fashion to functional programming, the specification of a computation is composed of a collection of such predicates and their associated clauses. The role of the outermost function application, whose value is assessed in a functional system, is now played by the outermost predicate together with its arguments. Given fixed arguments, the interpretation is similar - "execution" consists of deciding whether the predicate is true given the arguments and the associated definitions. However it is also possible to specify the outermost predicate with unbound arguments. The purpose of execution is now to find bindings to the arguments which allow the predicate to be satisfied, or to determine that no such bindings exist. There may be more than one such set of bindings and in a language of pure Horn clauses choice between these is non-deterministic.

At an abstract level, the process of evaluation may be seen as expanding and searching a tree of possibilities presented by consideration of the various dependencies between appropriate predicates and clauses. The process of expanding the tree by matching left hand sides of predicates to their "body" definitions (cf. expanding functions) is known as "unification". At the root of the tree are the outermost predicate and its arguments. Each of its sons correspond to one of the clauses whose disjunction defines the value of the predicate. Sons of the clauses correspond to the predicates whose conjunction defines the clause. Following a particular path down the tree corresponds to testing the validity of a certain collection of clauses and predicates under some specific bindings to some or all of the variables. Clearly parallelism may be introduced into the expansion and evaluation of the various sub-trees of nodes. However an important distinction exists between nodes corresponding to predicates and those corresponding to clauses.
Chapter 1. Generating and Controlling Parallelism

With the former, the satisfiability of the parent node is guaranteed by that of any of its sons. Thus a parallel evaluation can accept the bindings produced by any satisfied sub-tree as satisfying the parent goal. Exploitation of this is known as OR-parallelism. Satisfiability of the latter type of node requires the satisfiability of all of its sons. If these are explored independently in parallel (a process known as AND-parallelism) then the bindings produced to satisfy the sons may be mutually inconsistent. Checking for this occurrence may be complicated and could outweigh any benefits gained by the introduction of parallelism. For this reason pure AND-parallelism is typically not used in real implementations and work is focussed on the exploitation of OR-parallelism.

The programming language Prolog does away with the non-determinism of pure Horn clauses by specifying that clauses for some predicate should be investigated in textual order. This feature is used by Prolog programmers to influence the efficiency of sequential evaluation. Various other features have been introduced to different versions of Prolog, all with the intention of allowing the programmer to guide the abstract machine towards an efficient parallel implementation. Gross abuse of these facilities typically leads to programs which rely on knowledge of their implications (in terms of the implementation) to preserve correctness. This flies in the face of one of the original motivations for logic (and functional) programming, namely the separation of semantics from implementation. The various ins and outs of these amendments are not discussed here.

Given the preceding summary of some idealistic machines, it is now appropriate to examine proposed realistic parallel implementations with reference to the central issues of parallelism introduced at the start of the section.

In functional languages the problem of partitioning to produce parallelism is trivial. Parallelism is presented implicitly in the abstract execution models. In a string substitution scheme it is possible to proceed at multiple points within the
overall expansion (e.g. [41]) and in the more common graph reduction schemes [15,31], with or without combinators, the graph may be expanded and reduced at several nodes concurrently. The crunch comes when realistic distribution of this work is considered. With string manipulation the expression may easily expand in an irregular fashion and require re-organisation of its original distribution in store. Similarly, in graph reduction, the structure of the graphs produced is entirely specific to each problem instance. Furthermore, the structure of the graphs becomes apparent and evolves dynamically during execution. Thus any mapping scheme which tries to distribute the graph and the associated work-load effectively must be both dynamic and general purpose. There are two approaches to this problem. The first [31] attempts to balance work dynamically in a localised manner by allowing idle processors to grab work (effectively portions of the expanding graph) from busy neighbours. Contrastingly, schemes such as [15] take a more global view. The graph is stored as a globally accessible “pool of packets” which in practice is distributed across the local processor memories. An interconnection network deals with accesses to non-local packets. Clearly there is a trade-off here between the locality of access and lack of global scheduling of the former method, and the more complicated global access and distribution of the latter.

The distinction between “data” and “code” in an functional system is blurred – both are intermingled in the graph or string during expansion, reflecting their common foundations in lambda calculus. However, it is possible to recognize that the function definitions play a more static role, in some ways similar to that of more conventional code. They provide templates describing the expansions and reductions which may be applied to the graph. This is equally true when these definitions have been “compiled” into combinator form. Most systems [11,15,31] provide each processing element with a copy of all the function definitions which may be used during execution to manipulate independent areas of the graph or
string concurrently. There are problems presented by the property of referential transparency, particularly the implied copying of large "data-structures", which are important here. As with the example of Prolog's add-on features, some attacks on this problem introduce further annotations enabling the programmer to influence the execution mechanism [12]. These can have extremely undesirable effects. For example, the reduction machine of [66] as reported in [65] includes operators which "can, if not used carefully, violate the referential transparency property of reduction machines".

A functional program contains no explicit notions of communication or synchronization. However, in a realistic implementation these are introduced as a by-product of two of the features discussed above. These are the distribution of the program graph over local memories with resultant non-local accesses caused by edges connecting nodes residing in distinct memories, and any attempts at dynamic load balancing with all the message passing and checking involved therein.

In the data-flow approach the issue of problem partitioning is even simpler. A process similar to compilation translates the "high-level" program into a corresponding data-flow graph which uncovers all the potential concurrency between operators. Together with tokens representing input data required to get the computation started, this is the only "code" present. There is nothing analogous to the function and predicate definitions of functional and logic language schemes. Consequently all the problems of distribution, communication and synchronization are associated with the data-flow graph and the interactions between its node operators. Although the structure of the graph is static, it will only be apparent during (or even after) execution that some sections of the graph were more active than others.

Again, there are essentially two approaches to the graph distribution problem. In the first [18], all operator node storage and matching is isolated from
Chapter 1. Generating and Controlling Parallelism

the processing elements. There is no attempt to localise sections of the graph to particular processors or clusters and ready instructions are sent for execution to any processor. This is comparable with the global approach to load sharing in functional systems. Other data-flow machines [6,24] associate operator nodes and their token matching with particular processing elements. Result data tokens must therefore be routed to particular processors. This represents an attempt to localise communication at the possible expense of more balanced load sharing. Of course, it will still be necessary to route some packets non-locally. Consequently, it doesn’t avoid the problems concerned with providing what amounts to an arbitrary message passing scheme between large numbers of elements across a network.

Not surprisingly, the position with logic languages is close to that of functional languages. Although the logic approach has attracted less attention until very recently, the principles are similar. Problem partitioning to uncover parallelism is simple – the expanding search tree provides opportunities at every branch (bearing in mind the problems associated with AND as opposed to OR parallelism). Once again, the problem of distribution is not so easy – the structure of the tree only becomes apparent as execution proceeds. Communication is involved both in returning results and provisional bindings from sub-trees and in co-ordinating the processors into effective searching of appropriate areas. The role of the function definition is now played by the predicates and typical systems will allocate each processor a copy of all of these “templates”, allowing unification to proceed concurrently in distinct portions of the tree. [62] illustrates an approach exploiting only OR-parallelism in which idle processors may request work of other busy processors via a global interconnection network. The PIM-R machine discussed in [51] presents a “reduction” approach with each processing element having copies of all the templates and active nodes being shared across the processors as the computation progresses. The similarities between implementing functional and
logical languages are emphasised in [16] which discusses methods for executing Prolog on the Alice machine, originally designed for the execution of functional languages.

1.2.2 Idealised Parallel Machines

The systems discussed in the first category were characterised by the isolation of the abstract (or idealised) design space from the parallel, distributed implementation. The second category considers machines in which the two levels are closer together and in particular, those in which the user's design space includes explicit parallelism. The user is now responsible for partitioning the solution into a collection of concurrent processes. The abstraction remaining is concerned solely with the mechanisms by which such processes cooperate, and may take one of two closely related forms.

In the first, all processes are presented with equal access to some kind of shared memory space. In its loosest form, any process may attempt to access any item at any time. A variety of refinements of the scheme exist, each with its own rules defining the semantics which resolve or forbid clashing access requests. In a typical model, each such access is assumed to take unit time. The implementation task is to devise a scheme which can satisfy any legal combination of requests within some reasonable time on the realistic machine.

The second flavour of idealised machine discards shared memory based cooperation in favour of some form of explicit message passing. Typically each process is given the power to pass a message to any other in unit time concurrently with a collection of other such exchanges (i.e. a complete communication network is assumed to exist). Once again the implementation task is to mimic such behaviour in reasonable time. We now present a variety of machines from this category, both proposed and implemented.
Much existing work centres on the topic of creating efficient realistic simulations of a class of idealised parallel machines known as parallel RAMs (PRAMs) (see [55,74] for surveys). Such machines consist of a collection of \( n \) standard uni-processors each programmed independently and with its own local memory. Processors are assumed to proceed synchronously (i.e. every processor completes its \( i^{th} \) instruction before any begins its \((i + 1)^{st}\)). Additionally, they have access to an \( m \) location shared memory (SM), upon which they may operate during any instruction, with a variety of rules governing conflicts. In the most restrictive case, exclusive-read exclusive-write (EREW), no two processors may attempt to access the same shared memory location during the same time step\(^2\). Concurrent-read exclusive-write (CREW) PRAM machines allow an arbitrary number of read accesses to the same location but require writing to be unique, while CRCW PRAMs allow completely arbitrary access to any locations during the same time step. Again, a variety of schemes govern the semantics of clashes, ranging from those in which a write clash results in a random (i.e. junk) value being recorded, through those in which a randomly selected processor is successful and on to those in which the successful processor is chosen deterministically (e.g. by highest processor identity or smallest written value). The result of a read coinciding with several writes can be similarly defined. Some models (e.g. [73]) augment the PRAM instruction set to include operations such as “Fetch and Add”. Concurrent execution of several such operations on some location \( l \) results in each participating processor receiving a copy of the original contents of \( l \), while \( l \) itself has its contents incremented by the sum of all the values specified in the instructions. Clashes involving such new operations and old ones are resolved or prohibited by further rules.

\(^2\)It is not clear how such a scheme could be enforced upon the programmer in practice.
Proposed implementations of such machines associate each PRAM processor with a real sequential processor and its own local memory. The real processors are typically connected by some sparse network with unit time fixed length message passing between immediate neighbours only. Any other communications are forwarded through the network to their destinations. Since in reality there is no centralised memory, the idealised SM locations must be distributed across the local memories. Processors are required to cooperate in the simulation of the access patterns specified by each PRAM step.

Straightforward solutions map each PRAM SM location to a single local memory location. Thus, simulation of a particular memory access requires message passing between the active processor and the one responsible for the location. Realisation of a complete PRAM step is therefore reducible to the problem of concurrently routing a set of such messages around the network. This is closely related to the issue of sorting on networks and a variety of routing schemes have emerged, each with particular networks in mind. Useful results are obtained by the introduction of hashing techniques [47] to generate the shared to local memory mapping and of randomising routing phases [71] intended to alleviate the problems associated with awkward message permutations. In spite of good average or expected behaviour, any such scheme is vulnerable to PRAM steps in which all processors attempt to access locations stored by the same real processor. These can occur whenever \( m \geq n^2 \), and for fixed degree networks immediately imply a \( \Omega(n) \) simulation slow down, a performance level which could be matched by a single processor\(^3\).

A proposed remedy to this problem [47] is to keep multiple copies of each SM location in different real processor memories and to access only the most conve-

---

\(^3\)This is true even of EREW PRAMs, since it is not necessary that the \( n \) requests refer to the same location but merely to the same processor's local memory.
nient. Initially this was proved successful only for dealing with multiple reads. However, in a striking paper Upfal and Wigderson [69] show that the technique can be extended to include concurrent write steps. They present an algorithm which allows $T$ steps of an $n$ processor CRCW PRAM to be simulated by an $n$ processor constant degree, fixed network machine \(^4\) in $O \left( T (\log n)^2 \log \log n \right)$ steps assuming that $m$ is polynomial in $n$. The result has since been improved to $O \left( T \log^2 n \right)$ [5].

The method centres on the simulation of each SM location by $2c - 1$ real locations, where $c = \lceil \log_2 m \rceil$. In addition to its contents, each location notes a "time-stamp" indicating the simulated step during which its value was last updated. To write an SM location it suffices that a processor succeeds in writing at least $c$ (i.e. a majority) of its copies, also stamping them with the current time. To read an SM location a processor must also access $c$ of the copies and will accept the copy with the most recent time-stamp. As a result of the writing rule this must be one of the most recently updated copies and therefore correct.

Several projects have drawn from this pool of ideas in the implementation of actual machines.

The New York University Ultracomputer [22] presents the user with an $n$ processor CRCW shared memory parallel computer in which the processor instruction set includes the "fetch and add" instruction described above. The model is not a PRAM since no guarantee is made that processors will proceed synchronously. The physical machine consists of a set of $n$ processing elements which are connected to $n$ independent memory modules by an "omega" interconnection network [40] of depth $\log_2 n$. This can be seen to be very similar to the PRAM

\(^4\)In fact, the main thrust of the paper considers simulation of an EREW PRAM on a complete network of processors, and refers to existing techniques to extend the result to the more interesting models.
simulation schemes by imagining each processor to be connected to an unused memory module and each memory module to an unused processor. The use of a one-one mapping between physical and virtual SM locations means that the machine runs into the $\Omega(n)$ slow down problem. However, the authors assert that use of a suitable hashing function to define the mapping results in good average behaviour.

Similar principles were used to design the Heterogeneous Element Processor (HEP) [36]. Again, the user’s model is a collection of asynchronous processes accessing a shared memory. The PRAM notion of lock step progression through time is discarded in favour of explicit primitives for synchronizing processes during execution. Once again, the physical machine consists of a collection of processors connected to a set of memory modules by a sparse interconnection network, with a one-one memory mapping. The processing elements themselves are more complicated, each consisting of a process instruction pipeline containing several functional units. This allows the pipelined execution of several independent instructions from different processes to proceed concurrently. The aim is to hide the memory latency (the time spent waiting for the network to satisfy memory references) by executing other instructions in the meantime. Thus the network is also, in effect, pipelined. The ultimate goal is the achievement of 100% processor utilisation. However, this is obtained at the expense of deliberate under exploitation of the parallelism available in the program. Meanwhile, the concurrent routing of several sets of memory accesses per processor increases the likelihood of an undesirable distribution of requests across the memory modules.

The Linda model [3] presents an interesting adaptation of the traditional view of shared memory. A Linda memory consists of a set of tuples, each containing an ordered collection of data values. Tuples may be of varying length and are content addressable. A Linda program consists of a collection of independent asynchronous processes. These may access the tuple memory by means of three
primitive operations, "out", "in" and "read". The operation out(t) causes the tuple t to be placed in the tuple memory. The operation in(t) causes a tuple matching the template t to be removed from memory (if one can be found) with its fields passed to the process. The template can be left unspecified to varying degrees. For example, a process might request any five integer tuple whose first field contains 16. The operation read(t) is similar but does not cause the tuple to be removed from memory. Choice between a collection of suitable tuples is made non-deterministically (from the user's viewpoint), while failure to find any suitable tuple causes the requesting process to be suspended until one becomes available.

The design space is thus completely independent of the hardware and several implementations are reported to be under construction, including bus and hypercube based systems. The bus based system requires each processor to store a copy of the entire tuple memory in an attempt to cut down on bus congestion at the expense of increased storage. The hope is that a significant proportion of accesses will be "reads" requiring no communication, or "outs" requiring only one message broadcast each. On the other hand, the hypercube implementation assigns a unique node memory location to each tuple and relies on good routing and synchronisation algorithms to avoid congestion.

The abstract machines discussed above have emphasised the use of a collection of concurrent sequential processes accessing a static shared memory. In contrast, the user of the Connection Machine [26] is encouraged to think of it as an ordinary sequential host processor connected to an "active" memory. Memory elements are accessible to the host in normal random access fashion. Additionally, every memory location or object can be considered to be coupled to its own processing element. These are presented as being connected by a complete communication network, and operate synchronously under direct control of the host. The intended programming style is suggested by the accompanying C* language
Chapter 1. Generating and Controlling Parallelism

[56], an extension of standard C, run on the host machine. This allows operations to be specified as acting on individual memory locations (as in standard C) or upon the set of locations or objects satisfying some criteria (belonging to the same structure class, for example). In this way, all objects of a certain class may alter the contents of some other objects to which they have pointers. The usual variety of rules control the result of several such operations converging upon the same location.

In the implementation, writing via a pointer reference corresponds directly to passing a message between the two processes responsible for the objects. In practice, several such processes will be mapped onto the same processor and the complete virtual connection network is simulated by suitable routing algorithms on the underlying hypercube network which connects clusters of processors. In contrast to the machines discussed previously, the processors are now controlled by the host which broadcasts their instructions (in SIMD fashion). This emphasises the fact that the C* program is being run sequentially on the host with augmented instructions allowing the concurrent memory manipulations. The branching capabilities and block structure of C mean that several processors (or memory objects to the user) can be left waiting while some inappropriate block is executed elsewhere. The run time system on the host is responsible for ensuring that all such loose ends are tied up. Connection machine Lisp [75], a dialect of common Lisp, presents an alternative syntactic sugaring based upon the same foundations. It shares the same goal of encouraging a "data parallel" [27] (as opposed to "control parallel") programming style.

The problematic design task for the Connection Machine concerns the implementation of good routing algorithms for average and worst case patterns of access. Also, the hope is that the number of processes kept idle by branches in the code will not prove excessive. The latter point is closely related to the nature of the applications involved. Since C* includes C as a subset, it is easy to
write programs which make no use of parallelism whatsoever. On the other hand, suitable applications can be programmed to exploit it very effectively.

The examples considered so far have concentrated on some form of idealised shared memory space as the medium for inter-process cooperation. The Occam language [34] (based on CSP [28]) takes a contrasting standpoint which bases cooperation on explicit process to process message passing, with no shared memory. Although theoretically close to the shared memory model, the point-to-point communication scheme encourages a different approach to problem solving. An Occam process may be connected to any number of others by one way communication channels. A process runs asynchronously, accessing only its own exclusively local memory until it wishes to communicate with another process. Communication of some message down a channel takes place only when both sending and receiving processes indicate readiness (by attempting to execute output and input statements respectively). The sending process specifies the message, while the receiving process indicates the location in its memory where the new value should be stored. In this manner, explicit synchronisation of processes can be achieved. Write clashes are explicitly prohibited by a syntactically enforced rule which forbids processes running concurrently to write to common shared variables.

The Occam programmer is required to consider the partitioning of data into independent sets in addition to the partitioning of work into processes. Any sharing of data must be programmed explicitly by message passing. In theory, Occam programs allow any complexity of process connection network, including complete interconnection. However, existing systems provide no support for

---

5Note that a complete Occam connection network is more powerful than that used in [69]. In the former any number of processors may successfully communicate with the same process in unit time. Only one message may be sent or received per time step in the latter.
automatic mapping onto available realistic networks. Consequently, Occam tends to be used for the explicit programming of real hardware rather than for specifying idealised algorithms.

The Cosmic Cube [59] presents a design space similar to that of Occam. A collection of strictly sequential processes are executed concurrently and communicate by means of a complete interconnection network. In contrast to Occam, a message passed does not imply a synchronisation between the two acting processes. Messages are buffered and additional processes can explicitly check whether any messages have been received. Each process can send at most one message during each time step.

In terms of handling the key issues in the use of parallelism introduced previously, the machines in this category exhibit a clear shift in responsibility from the system to the user. Thus, the user is now entirely responsible for problem partitioning, by specifying the processes which may operate concurrently. There is a split over the issue of data sharing. In the shared memory abstractions all data is shareable and the system takes responsibility for implementation. In those without shared memory, the user is required to partition data between processes and handle sharing explicitly by message passing. Code sharing is not an issue, while only the non-shared memory machines provide facilities for inter-process communication. The burden here is still carried by the systems which provide abstractions to more general networks than appear in the hardware.

Finally, a variety of approaches are evident to the issue of synchronisation. At one extreme, the Connection Machine's straightforward progression through an essentially sequential program relieves the user of any difficulty. The system handles the problem of bringing together divergent threads introduced by data dependent branching. The Occam model requires any synchronisation to be expressed explicitly but the semantics of its communication primitives make this
Chapter 1. Generating and Controlling Parallelism

relatively simple. Shared memory models are the least helpful here, requiring careful use of shared variables to ensure process rendezvous, though the lockstep progression of the strict PRAM models and augmented instructions are useful.

1.2.3 Low Level & Restrictive Abstractions

The final category completes the journey through the levels of abstraction afforded by parallel systems. It includes machines in which the user is required to consider both explicit parallelism and the realities of the hardware topology in designing a solution.

The Meiko Computing Surface [46] is a typical example. Computing power is provided by a large collection of independently programmable processing elements, each with private local memory and four bi-directional communication channels. All channels are linked to a central switch where they may be connected in pairs to realise any network of degree four between the processors, as specified by the user. An Occam solution to the problem in hand is constructed and the user is responsible for the explicit mapping of Occam processes to processors and channels to actual communication links. Thus, while communication between processes sharing the same processor is unlimited, actual parallelism between them is non-existent. On the other hand, communication between genuinely concurrent processes (on distinct processors) is restricted by the selected topology. The freedom (and responsibility) for the specification of both hardware and software places the Computing Surface unreservedly in the “do-it-yourself” school of parallelism.

Snyder’s “Chip” machine [60] and the accompanying “Poker” [61] programming environment are in similar vein. The Chip hardware presents the user with a two dimensional grid of processing elements, each with local memory. Each row and column of $p$ processors is separated from the next by a row or column of
Chapter 1. Generating and Controlling Parallelism

2$p$ switches, one between the "faces" of each pair of adjacent processors and one between each set of four adjacent processor "corners". Each switch can be set to implement direct connections between non-intersecting pairs of neighbouring elements (whether processors or other switches). Some of the possibilities (including paths crossing at a switch) are illustrated in figure 1–1, where squares represent processors and circles represent switches. Thus the realisable physical communication networks are restricted to those which can be mapped onto this structure, and in particular to a maximum degree of eight. The Poker environment (running on a sequential host processor) allows the user to specify the required connection network (given the restrictions) and a mapping of sequential processes and communication ports onto processors. A succession of communication networks may be incorporated into the same program by resetting the switches during execution. The FPS T-series [48] machine offers a similar model, but with no flexibility in the hardware configuration. Processors are connected into a binary hypercube onto which users are required to map Occam processes explicitly.

The systems considered so far in this category share the property of placing responsibility for the description and implementation of parallelism entirely in the hands of the user. As a direct consequence of offering no assistance, they
pose no associated problems to the system designer. Thus there are few points of interest in the context of this discussion.

In contrast, the final approach considered reverts to the format of letting the system do the work in extracting parallelism. Although this would seem to imply its inclusion in the first category, its presence here is justified by the fact that good performance is only achieved in situations so specific that the user must in fact, be fully aware of the underlying details. The technique in question is the use of "vectorizing" compilers to generate code for "pipelined processors". The design space for such a system is simply a straightforward sequential language (typically Fortran). The task of the compiler is to take such a program and isolate (or perform simple statement re-arrangements to generate) long sequences of instructions performing the same arithmetic operation upon streams of operands. These streams can be considered to be vectors (hence the name) and can take advantage of special purpose arithmetic hardware units. These extend the standard techniques for pipelined instruction fetch and decode, common in ordinary sequential machines, to include several (typically 5–10) stages in the execution of simple arithmetic operations. A full \( k \)-stage pipeline contains \( k \) instructions at various stages of completion and can produce results at up to \( k \) times the rate of similar, but non-pipelined, hardware. In the same way, \( p \) full \( k \)-stage pipelines can give a \( pk \)-fold improvement. Not surprisingly, the major implementation problems are concerned with the task of keeping the pipelines full. Compilers must consider the data dependencies between statements and more particularly between iterations of loops, in order to identify sequences of operations which can be packaged up as vectors without destroying program correctness. Since a complete analysis rapidly becomes infeasible, the search is usually restricted to certain well understood cases. Thus a user must be well aware of the type of programs which will be successfully "vectorized" in order to ensure satisfactory utilization of the hardware. User responsibility is emphasised in some systems by
the inclusion of explicit vector operations in the sequential language. [52] presents a thorough discussion of the compiler techniques involved in the area, while [32] includes lengthy analyses of several pipelined machines.

Turning once more to the identified central issues, it is clear that the first group of machines in the category represent a completion of the transition from system to user responsibility. Code sharing is not relevant, and partitioning, distribution, data sharing and communication are all entirely in the hands of the programmer. Some support for synchronisation is provide by the machines which run Occam code on each node.

Judged on these criteria, the vector and pipeline approach may appear to be misplaced in this category. Since the architectures involved are typically not really distributed in the sense used previously, the usual issues are not wholly appropriate. It can only be emphasised that their inclusion here is justified on the grounds of their restrictiveness and specialisation.

1.3 Conclusions

The work presented in the preceding survey addresses the problem of providing a useful, general purpose abstract machine, with an implementation efficiently harnessing the computational power of parallel hardware. Consideration of the examples illuminates two courses of action, sharing one common principle. All of the projects present the abstract machine as a general purpose programming facility, whether sequential or parallel, imperative or non-procedural, entirely independent of application. Divergence appears over the level to which the tools made available are independent of the hardware. Although varying in other respects, machines in the first two categories are clearly designed with hardware independence in mind. They could be implemented on any particular hardware
configuration with no visible difference to the user except in performance. Machines in the final category are clearly targeted at specific hardware and may even make a feature of explicit hardware manipulation. Both approaches have advantages and drawbacks.

The highly abstract machines of the first category have the obvious advantage that the user specifies solutions in familiar, well understood languages, free of concurrency. Against this must be weighed the difficulties of implementation and the corresponding loss of power over that available from the bare hardware. The fact that the degree of exploitable parallelism is dependent upon the program and is variable throughout execution makes problem independent analysis extremely difficult. Existing experimental results (e.g. [43]) report on only small scale machines. It is far from clear how well they can be expected to scale up to thousands or more processing elements.

Algorithms used to implement the idealistic parallel machines of the second category are far more amenable to traditional analyses of time complexity. The algorithms usually involving routing and sorting across networks and it is often possible to estimate worst case performance to within a constant factor. Often an algorithm's complexity will be independent of the details of the particular step being simulated. In these cases it is possible to provide accurate estimates of the run time for a complete user program by multiplying the number of idealised steps by the simulation slow-down per step. Thus, in return for the effort of specifying an explicitly parallel solution, the user is rewarded with a more concrete estimate of run time. On the other hand, the standard overhead is incurred whether or not a particular program uses the full power of the simulation.

\textsuperscript{6}Of course, in theory these could also be simulated on any hardware. However, this would contradict their original motivation.
Chapter 1. Generating and Controlling Parallelism

Contrastingly, the explicitly parallel systems in the third category incur only constant overheads, since there is no significant implementation cost. Programs are mapped directly onto the hardware by the user, performing as expected, and a cunning solution can extract the maximum achievable performance. In practice there are many examples of processor networks whose topology is especially well suited to the structure of certain problems. One of the most striking (and well known) is the relationship between shuffle networks and fast fourier transformation [63]. Similarly, all the automated work in vectorisation is done by the compiler, with the code produced running directly on the hardware without further intervention. In suitable cases (see [32] for a lengthy discussion) this can also be very efficient. However, in all but the simplest cases, construction of such a solution requires considerable effort and understanding of the system and the search may be fruitless. There is a clear parallel here with direct use of machine code in conventional systems.

In summary, existing systems either present a relatively friendly abstraction with hidden, unavoidable overheads (the first category), a harder to use abstraction still with in-built but better understood overheads (the second category) or a low-level, awkward abstraction without overheads. The uniting feature is that all cases present a single, general purpose interface to the user.

In this thesis, an alternative style of abstraction is proposed, in which the notion of a single universal interface is rejected. Instead, the user is presented with a selection of more specific solution "skeletons", each suited to a particular style of algorithm description. The approach is motivated by the hope that the selection will be wide enough to make the abstract machine reasonably general purpose, while having implementations of each individual skeleton which make efficient use of the hardware. The proposal is discussed in more detail in chapter 2, while subsequent chapters describe a particular set of skeletons and their implementation on a particular parallel machine.
Chapter 2

Algorithmic Skeletons – A New Approach

In this chapter, we propose a new approach to the problem of designing a system which transforms the speed of parallel hardware into a more useful form, based on the idea of “algorithmic skeletons”. The first section presents the abstract machine, while the second discusses the choice of the particular model of parallel hardware used throughout the remainder of the thesis. The final section describes the principles and methods underpinning the subsequent discussions on implementation.

2.1 The Abstract Machine

It was noted in section 1.3 that the new abstract machine discards the concept of a universal interface which is common throughout conventional programming languages. In contrast, the abstraction proposed here presents the user with a selection of “algorithmic skeletons”. Each skeleton captures the essential structure of some particular problem solving style or technique. To solve the problem in
Chapter 2. *Algorithmic Skeletons – A New Approach*

hand with this new system, the user is required to select a skeleton which describes a suitable method of attack. This must be fleshed out with procedures and data-structures customising it to the specific problem. As will be shown, each instance of these procedures will be executed purely sequentially and may therefore be presented in any convenient programming language. A particular choice is of no interest here and will not be discussed further.

The proposal may be clarified by imagining an interactive session with the type of "programming environment" with which the abstract machine could be presented in practice. In similar style to Poker [61], this would reside on some front-end workstation. Upon activating the system, the user is presented with a menu listing the available skeletons. Just as with a conventional programming language, details and examples of these would be described in the accompanying user's guide. An appropriate skeleton for the problem in hand is selected. Similarly, the user selects the language in which the procedures and data structures fleshing out the skeleton will be specified. As noted above, instantiations of such routines will be executed purely sequentially on independent processors – thus the choice of languages is restricted only by the availability of standard compilers.

The system responds by displaying the generic program describing the operation (at an abstract level) of the selected skeleton, illustrating the way in which the as yet unspecified routines will interact to produce a solution. For example, the recursive divide and conquer skeleton of chapter 3 has the abstract specification

\[
F(I) = \text{if} \ \text{indivisible}(I) \ \text{then} \ F(I) \\
\text{else} \ \text{Join}_k (\text{map} (F, (\text{split}_k(I))))
\]

(the precise meaning will be made clear in chapter 3 itself). Finally the system prompts the user to provide descriptions, in the selected language, of the data structures and procedures required to turn the generic skeleton into a complete solution specification. Provided that these are consistent in terms of the selected
language syntax, the "programming" task is now complete. To initiate a "run" of the "program" it only remains to indicate the location (in the local file system) of a data structure describing an instance of the problem, just as the name of a Pascal "file of records" might be specified conventionally.

At this point the system takes over. The data and compiled code fragments are loaded across the parallel hardware together with the system code required to implement the selected skeleton (this may already be present if the local memory is large enough). The hardware executes the skeleton and returns an instance of the appropriate data structure (representing the results) to the file system. The mechanisms implementing this process are entirely hidden from the user.

It is worth emphasising that the selection of a particular skeleton ties down the entire solution process. There is no inter-play between skeletons of the type existing between constructs of a conventional programming language. This gives the abstract machine its characteristic non-uniformity. Its goal is to attain the property of being "general-purpose" by providing a wide enough range of individually specialised options, rather than a single, all-purpose structure. The motivation for this choice stems from the survey of chapter 1 and is discussed in section 2.3.

Equally, it is important to note that the system described above neither required nor permitted mention of any details of the hardware (not even that it was parallel in any way). This is consistent with the principle of maintaining a clean abstraction between idealised and actual machines.

A useful analogy can be drawn between the "routine library" facility, often found as part of a conventional abstract machine, and the notion of skeletons. With the former, a collection of very specific (at least when compared with the skeletons) collection of program modules is provided. An appropriate selection of routines is made, and conventional language constructs are used to build these into a framework which solves the problem. The user is required to conceive of and
describe the overall solution, but is given help with the details. In contrast, the skeletal machine presents a collection of ready made (and invisibly implemented in parallel) frameworks. The user’s task is to select one and fill in the details.

It was hinted in chapter 1 that the provision of explicit parallelism in the abstract machine is not, in itself, something of merit. Consequently, it would seem to be desirable that skeletons should be presented, at this level, as operating without parallelism. Three of the four skeletons presented in chapters 3 to 6 satisfy this goal entirely. Although the skeleton of chapter 4 could be presented without reference to concurrency, its properties are probably most easily understood (and certainly described) in terms of a parallel implementation, albeit in an idealised form.

### 2.2 Parallel Hardware

Before considering an implementation of the abstract machine, it is necessary to specify the underlying parallel hardware. It is important to note that the abstract machine has been described in a completely hardware independent way. Thus, selection of a different model of hardware would lead to a different implementation of the same skeletons.

As noted in section 1.1, the recent technological drive has been towards providing large scale replication of processing and storage elements, sharing the same design and fabrication processes. It is essential that the magnitude of these advances is not lost upon the system designer. Parallel implementations of abstract machines should be designed to make effective use of essentially indefinitely expandable numbers of significantly powerful components. Of course, any real machine sitting in the corner of a room is most definitely of a fixed, finite size. However, it is vital that machines be designed in such a way that enormous hard-
ware expansion can be accommodated within a uniform framework. To design a highly efficient ten processor machine which grinds to a halt when expanded to a hundred processors is to disregard the real significance of the new hardware. For this reason, performance analyses of the asymptotic (in the number of processors) type employed in this thesis must initially carry more weight than experimental or simulated results derived from small scale implementations. It is most important to get the broad principles right first and to worry about the details later.

The range of existing and proposed parallel hardware is diverse (e.g. see [32]). However, to provide coverage of sufficient depth for the purposes of this thesis, it is possible to isolate the relationships between three types of hardware object as being of prime importance. These are the processing elements, the memory modules and the interconnection network.

The issue relating processing elements to memory modules concerns the question of whether particular elements are associated physically with particular modules. For example, the Computing Surface [46] is constructed from a collection of tightly coupled processor-memory pairs. Each processor has exclusive access to its own memory module at the hardware level – non-local access must be arranged by message passing in software. On the other hand, processors in [36] are independent of any particular memory module. Access capability, via the network, is equally distributed. This equality is both a strength and a weakness. By removing the problems associated with arranging access to “distant” data it also prohibits any super-imposed system from exploiting locality to improve performance. The hardware selected for the “skeleton” machine takes the approach of associating each processor with a unique memory module. As suggested, an attempt is made to exploit this locality in the implementation.

The other important area concerns the network structure and its degree of integration with processors and memories. In terms of structure, a wide variety
of networks have been considered previously, ranging from the simplest "bus" scheme, in which every element competes for access to the single communication link, to complete interconnection schemes in which every element has a distinct connection to every other. Between these extremes range rings, grids, trees, shuffles, hypercubes etc. (again, see [32]) for a survey).

The issue concerning the integration of the network with processors and memories is similar to that relating these two items themselves. Should processors and memory be distributed throughout the network and active within it, or should they sit beside it, feeding in requests and receiving replies?

Again, the choice essentially concerns the presence or absence of a notion of locality and once more a decision is made in favour of its presence, for the purposes of this thesis. Although we cannot hope to arrange for every memory access to refer exclusively to local memory, it should be possible to exploit some degree of near-neighbour locality. This will only be possible if the notion of "neighbour" has any meaning and if the processors themselves influence the flow of data. Thus, in the selected hardware, processor-memory pairs will be located at the vertices of the selected network, actively forwarding non-local messages, requests and so on.

An immediate implication for the choice of a particular network is that the level of connectivity between physical elements must be sparse. It is not reasonable to expect a single component (e.g. a bus) to connect directly to a large number of other components and still provide reasonable performance. A wide variety of networks have been proposed as a basis for general purpose parallel computers ([42] provides a recent starting point for a survey), with consideration given to issues such as degree, diameter, optimal layout, wire length, regularity and fault-tolerance. The requirement for sparsity suggests that only networks whose degree grows as a small function (say $O(\log n)$) of the number of elements
Chapter 2. Algorithmic Skeletons – A New Approach

$n$, or even only those in which the degree is constant, should be allowed. Furthermore, the poor results for two dimensional layout, and consequently edge length, of most “logarithmic diameter” networks (e.g. as reported in [10,68]) suggest that networks based on hypercubes, shuffles, etc. may also be discounted (although the decision is less clear cut here). Of the regular networks with logarithmic diameter, only the tree appears to have a suitably concise layout (see chapter 3), but the potential bottleneck at the root and the long edges there count against it. Meanwhile, constant degree rings have too large diameter.

On the strength of these arguments, the square grid with its poorer than logarithmic diameter, but good degree, layout, wire length and regularity\(^1\) is selected as a foundation for the parallel hardware to be used in the subsequent discussion on implementation. Since the relative importance of the factors considered is open to debate, it is not suggested that this choice is the only one possible. Instead, it is simply claimed that the model is unarguably realistic and practical. Other networks mentioned could equally well be used, affecting only the implementation and not the abstract machine. In particular, it would be possible to exploit the results of [10] to take advantage of the added regularity of the “wrapped grid” produced by directly connecting opposing sides of the conventional grid.

Figure 2–1 shows a small instance (with 16 processors) of the selected hardware\(^2\). Each square represents a standard sequential processor with its own memory. Lines connecting adjacent processors should be interpreted as bi-directional communication links, capable of transmitting a single word in each direction between the local memories of neighbours in unit time. In the style of the Inmos trans-

---

\(^1\)And for that matter, fault-tolerance, although this is not considered here.

\(^2\)For convenience, subsequent illustrations shrink the connecting links to have zero length, and represent the hardware as a simple grid.
puter [33], it is assumed that the processor instruction set includes “send” and “receive” instructions which control the links, and that all four links may operate concurrently. None of the assumptions affect the asymptotic results obtained by more than a constant factor over those achievable with a more restricted model. Similarly, it can be assumed that all basic word instructions (from a typically “reasonable” instruction set) take unit time.

The communication links are assumed to operate the Occam model of synchronisation [34]. Exchange of a datum only takes place when sending and receiving processors are ready to execute “send” and “receive” instructions respectively. The first processor ready is forced to wait for its neighbour. Thus, no notion of a global clock is provided or needed. Although we would expect identical devices to proceed at essentially the same rate, strict lock step operation is not required. Again, for the purposes of asymptotic analysis, any such minor discrepancies in clock speed can be glossed over since the synchronisation mechanism is sufficient to ensure correctness\(^3\).

\(^3\)Although care must be taken to avoid deadlock.
In summary, the selected model of parallel hardware and the basic facilities which it provides may be seen to be entirely reasonable and realistic. There are no concealed tricks and the cost of any overheads is independent of the size of the machine considered, in terms of number of processors. For example, no unit time broadcast is assumed.

2.2.1 On Measuring Performance

In terms of performance, interest is focussed on the efficiency with which a large grid of processors can implement the system with respect to the performance of a single processor. Since the key issue is the introduction of large scale parallelism, it is important that more peripheral issues are not allowed to cloud the comparison. In particular, the machines comprising the square grid should be identical to the uniprocessor machine against which their performance will be considered. A brief discussion of the capabilities of such a machine and their implications is now appropriate.

The first important point to note is the implicit assumption that the machine has “enough” memory to implement any instance of a given skeleton. Any complications introduced by a failure to meet this assumption would apply equally to sequential and parallel cases and would therefore serve only to hide the real issue of parallelism. Secondly, the assumption of “unit time” operations from the basic instruction set implies the unwritten corollary that asymptotic results obtained from the model apply only when the manipulated values can be represented within the word length of the machine. Once more, this applies equally to parallel and sequential implementations and to include extra terms to cater for this would again obscure the real issue. Thus, in a sequential machine it is assumed that the retrieval of an integer from memory is a unit time operation, rather than a $\Theta (\log m)$ time operation with $m$ being the magnitude of the integer. Similarly,
it is assumed that the transfer of an integer between neighbouring processors is a unit time operation. Again, standard "fixes" to circumvent any real problems in practice would apply equally to sequential and parallel machines.

The second assumption requires one final note of caution which applies uniquely to parallel implementations. Some of the skeleton implementation algorithms to be presented make use of unique processor identifiers. At certain points these may be transferred between neighbours, using what is assumed to be a unit time operation. Strictly, $\log_2 p$ bits are required to distinguish unique identifiers in a $p$ processor machine and the unit time assumption is only valid when $\log_2 p \leq w$, the word length. For larger $p$, multiple transfers would be required increasing the time needed to $\Theta (\log p)$ in such cases. However, a typical $w$ of 32 or 64 allows $p \leq 2^{32}$ or $p \leq 2^{64}$ before this becomes an issue. Once again, the inclusion of such an addendum to relevant analyses would only complicate results in a way which would have little or no significance in practice – if a $2^{32}$ processor 32 bit machine is considered feasible, then a small extension of the word length would surely not pose any serious difficulty.

### 2.3 The Implementation

It has been emphasised in the previous two sections that the skeletons comprising the abstract machine are independent. Consequently, their implementations can also be considered separately, as indeed they are in chapters 3 to 6. This independence provides another of the motivations for the new approach. By concentrating on the efficient implementation of several distinct and relatively simple systems, it is hoped to achieve better overall performance than can be obtained by implementing one monolithic system designed to handle all possibilities.
Chapter 2. Algorithmic Skeletons – A New Approach

The implementation of each skeleton must ensure that the grid perform all the work described by the abstract specification. As well as the "real" computational tasks, this will involve dealing with the distribution of and access to non-local data, as implied by the distributed nature of the hardware. Fortunately, the problems associated with such a task are eased by the restricted nature of each individual skeleton – the pattern of work is clearer than for a more general system. Thus, each skeleton will have an associated controlling program, a copy of which is loaded (or given sufficient space, permanently sited) at each processor. Execution of each skeleton can be decomposed into a sequence of phases. The controlling program, given knowledge of its processor's absolute location in the grid, determines this sequence of actions required of the processor to execute that particular skeleton. For example, some data may be expected from one direction which must be manipulated and dispatched in another direction.

The format of the data and the nature of the manipulation will vary from problem to problem, but its arrival and departure constitute structural properties of the skeleton, applicable in all instances. Having loaded the controlling program, each processor must acquire a copy of the problem specific details and procedures as provided by the user. Since these are identical for each processor, it will be a simple task to arrange for copies of these to enter and sweep through the network. Similarly, the instance specific data is loaded to the local memories using the available external channels.

4The details of what lies beyond the grid are beyond the scope of this thesis. The loading process described here is of no interest, being entirely dependent upon the external I/O systems provided, and is comparable with the process of booting a conventional machine and loading the memory. The real problems, addressed in the bulk of the thesis, concern controlling the manipulation of the data to produce solutions efficiently.
At this point, the system is ready to begin execution of the skeleton by having each processor execute the appropriate control program. Quite simply, this consists of a series of calls of the user specified procedures operating sequentially on local data, punctuated by a sequence of communications required to ensure the correct distribution and exchange of such data between processors. As will become clear in subsequent chapters, such communications may take two forms, compulsory or optional.

In the former the arrival, manipulation and dispatch of data form an inherent part of the skeleton’s computational requirements. They can be guaranteed to occur in any problem instance using the skeleton. The synchronisation primitives described in section 2.2 are sufficient to be able to ensure that data transfer takes place only when both parties are ready and that it cannot be avoided. In the latter example, a communication may or may not take place at some particular point in the computation, depending upon instance specific data. However, it will be possible to reason that if no data arrives within a certain time period, that none will arrive during the phase. Such time periods are measured in terms of a number of local communication pulses. To ensure synchronisation, blank pulses will be filled by sending empty packets. In this way, pulses can be counted locally and suitable action taken. Note that this technique is only used when appropriate to a particular skeleton and not throughout the implementation of the whole machine, as might be required for a uniform general purpose abstraction. Finally, the transformed data is dumped to the external system.

This fragmented approach to providing a general purpose abstract machine with a concealed parallel implementation is apparently unique. Some recent work has considered parallel implementation of individual classes of algorithm [53,43, 49]. However, the concept of bringing several classes onto the same hardware together with a clean, hardware independent abstraction of each, while presenting the whole collection as a coherent machine, is new. The rest of this thesis is
Chapter 2. Algorithmic Skeletons – A New Approach

devoted to an investigation of possible implementations of four skeletons which
could form such a machine.

The abstractions presented by the skeletons of chapters 3, 4 and 5 each
emerged from the consideration of a selection of existing algorithms with and
without existing parallel implementations. The approach embodied by the “rec-
cursive divide and conquer” skeleton of chapter 3 is already widely recognised.
The “task queue” skeleton of chapter 4 is founded upon algorithms such as the
“shortest paths in graphs” method of [20] and the “packet pool” technique of
[15], while the “iterative combination” skeleton of chapter 5 was distilled from
the “minimum spanning tree” algorithm of [8] and the “PLA partitioning” tech-
nique proposed in [25]. The motivating algorithms as originally described were
designed with a widely varying selection of underlying machines in mind. Thus,
the thrust of the work presented in chapters 3, 4 and 5 has been to isolate and
abstract their underlying structure and to consider the automatic implementation
of these structures using the facilities provided by the grid machine.

By way of contrast, the “cluster” skeleton of chapter 6 represents the result
of an experiment in constructing a skeleton “from the hardware out” rather than
“from the abstraction in”. An abstraction and its implementation evolve from
consideration of the structural properties of the grid. Finally, chapter 7 reviews
the “skeletal machine”, considering its novel approach, implementation and pos-
sible future development in the context of the systems discussed in chapter 1.

2.3.1 A Note on Notation

The $O, \Omega, \Theta$ notation introduced by Knuth [38] is used throughout this thesis to
indicate the asymptotic magnitude of various quantities. Thus, the statement

$$f(n) = O(g(n))$$
Chapter 2. Algorithmic Skeletons – A New Approach

means that there exist positive constants \( C \) and \( n_0 \) with \(|f(n)| \leq Cg(n)\) for all \( n \geq n_0 \). Intuitively, and for our purposes more appropriately, this may be interpreted as meaning that \( f(n) \) “grows no faster than” \( g(n) \). Similarly,

\[
f(n) = \Omega(g(n))
\]

means that there exist positive constants \( C \) and \( n_0 \) with \(|f(n)| \geq Cg(n)\) for all \( n \geq n_0 \), which may be interpreted as meaning that \( f(n) \) “grows at least as fast as” \( g(n) \). Finally,

\[
f(n) = \Theta(g(n))
\]

is true if and only if \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \), with the interpretation that the two functions “grow at the same rate”.
Chapter 3

The Recursive Divide & Conquer Skeleton

3.1 Abstract Specification

The first algorithmic skeleton to be considered, "recursive divide and conquer" (RDC), is presented here\(^1\). This well known technique (e.g. see [2]) has been applied to a wide variety of problems and needs almost no introduction. It is used when a problem solution can be defined recursively as some function of a collection of smaller instances of the same problem, generated from the description of the original instance. Recursion is avoided if the particular instance is indivisible and is solved by some other non-recursive method.

The skeleton discussed here allows the user to specify solutions of this form with one additional restriction concerning the degree of recursion \(k\) (i.e. the number of "sub-instances" generated from some "super-instance"). For the skeleton

\(^{1}\)This chapter is a revised version of [14].
presented here this must be pre-determined, and constant through all levels of recursion. Furthermore, for reasons which will become clear in section 3.2.3, the results presented in this chapter will require \( k \) to be either 2 or a perfect square. However, the implementation technique itself will apply to any \( k \), with reduced efficiency as indicated. Thus, the skeleton is used to compute the solution of some instance \( I \) of a function \( F \) which may be described as

\[
F(I) = \begin{cases} 
  \text{if } \text{indivisible}(I) \text{ then } f(I) \\
  \text{else } \text{Join}_k(\text{map}(F,(\text{split}_k(I))))
\end{cases},
\]

where "indivisible" is a function which inspects a problem instance and decides whether it can be solved recursively, "\( f \)" is the straightforward function for solving indivisible (or "base case") instances, "\( \text{split}_k \)" is the function which returns the \( k \) sub-instances for recursive solution and "\( \text{join}_k \)" is the function which combines the solved sub-instances to solve the super-instance from which they originated. The term "map" is simply used for syntactic convenience and, in the usual sense, implies that \( F \) is to be applied to each instance produced by "\( \text{split}_k \)".

To use the skeleton, the user must specify the degree of recursion \( k \), provide a type description and details of the instance \( I \) and specify the functions "indivisible", "\( f \)", "\( \text{join}_k \)" and "\( \text{split}_k \)" which act upon objects of this type. These items can be specified in any language for which a compiler exists for the standard sequential grid processors.

It is worth noting that the skeleton can be used less restrictively than the fixed \( k \) might suggest since, in practice, the choice of \( k \) can be used to specify an upper bound on the degree of recursion. Instances which split into less than \( k \) parts may be supplemented by dummy, empty sub-instances. Of course, it is the user's responsibility to cater for this in the specification of the functions. As far as the implementation is concerned, each divisible instance will decompose into \( k \) sub-instances. On a more cautionary note the user must be aware that deliberate
over-estimation of $k$ will result in a disappointingly inefficient (although correct) implementation. Thus, this freedom should be used with care and only when strictly necessary.

The simplest and commonest RDC algorithms have $k = 2$. A typical example is the “mergesort” algorithm for sorting a list. Here, a list is indivisible if its length is 0 or 1, the base-case function “$f$” simply returns its argument, the “$split_2$” function divides the list into two halves and the “$join_2$” merges the two recursively sorted sub-lists.

The first implementation discussed in section 3.2 considers such binary RDC skeletons. However, it may often be more convenient (or even essential) to describe solutions in terms of larger $k$. For example, problems concerning multi-dimensional phenomena may be more suitably described with $k = 4$ or more. Thus, section 3.2 proceeds to introduce an implementation technique for more general $k$. The chapter is completed with an analysis of the performance which can be achieved, some proposals as to how this may be improved and the presentation of further examples.

### 3.2 Implementing the Skeleton

#### 3.2.1 An Idealised Implementation

The RDC skeleton, as presented in section 3.1 has an obvious parallel implementation on a tree of processors. Such implementations have been considered previously [53,30,43]. The root processor is presented with the whole problem instance which it sub-divides. Sub-instances are dispatched to its $k$ sons and recursion proceeds down the tree. Eventually, further sub-division is impossible and the leaf processors compute the base-case function of their respective instances,
passing the results back up the tree. Internal tree processors coalesce results using 
"join_k" until the root processor produces the solution to the whole problem. Subsequent sections propose and analyse solutions to the problems involved in moulding this idealised style of implementation to fit the grid.

3.2.2 The Two Way Split and Binary Trees

In order to simulate the idealised implementation on the grid, it is necessary to find some suitable mapping of tree processors to grid processors. Valiant [70] has shown that any v vertex tree of degree 3 or 4 can be embedded without shared edges into a grid of degree 4 using an asymptotically optimal $O(v)$ vertices. The "H-tree" (see figure 3–1) is a well-known layout of a complete binary tree on the grid with the advantage (for ease of implementation) of being very regular. In particular, all paths between edges at some particular level and their parents are of the same length. Although not all vertices of the grid are active in the embedding it can be shown that the H-tree layout of v vertices occupies a square grid of $\Theta(v)$
vertices. Therefore, associating each active H-tree processor with a processor from the idealised binary tree of section 3.2.1 results in an implementation of the RDC skeleton for which the number of inactive processors becomes negligibly small as \( v \) grows large. This is compatible with our principle of getting things right on a large scale, with expansion in mind.

Although most readily understood pictorially, the H-tree layout can be described quite concisely.

Suppose that the vertices of the grid are labelled with pairs of integers in the usual Euclidean fashion, with \((0,0)\) located at the centre. The root of the tree is mapped to grid vertex \((0,0)\). Then, recursively, for each tree vertex at depth \( d \), \( 0 \leq d \leq \log_2\left(\frac{v+1}{2}\right) - 1 \), mapped to grid vertex \((a,b)\), its two sons are mapped to grid vertices

- \( (a, b + 2^\frac{1}{2}\left(\log_2\left(\frac{v+1}{2}\right) - d - 2\right)) \) and \( (a, b - 2^\frac{1}{2}\left(\log_2\left(\frac{v+1}{2}\right) - d - 2\right)) \) if \( \log_2\left(\frac{v+1}{2}\right) \) and \( d \) are both even or both odd, or
- \( (a + 2^\frac{1}{2}\left(\log_2\left(\frac{v+1}{2}\right) - d - 1\right), b) \) and \( (a - 2^\frac{1}{2}\left(\log_2\left(\frac{v+1}{2}\right) - d - 1\right), b) \) if one of \( \log_2\left(\frac{v+1}{2}\right) \) and \( d \) is even and the other odd,

with connecting edges mapped through grid processors lying directly between those mapped to tree vertices at level \( d \) and those at level \( d+1 \). These intermediate processors each act as unit time delay wires, forwarding information.

It is simple to show that this layout is asymptotically optimal in terms of the size of grid used.

**Theorem 1** The complete H-tree layout of \( v \) vertices occupies a minimal bounding rectangle of \( \Theta(v) \) grid vertices.

**Proof:** Consider cases in which the smallest bounding rectangle of the H-tree is square (those in which the depth \( \log_2\left(\frac{v+1}{2}\right) \) is even). From the description of
the layout it can be seen that the maximum y co-ordinate (and by symmetry the maximum x co-ordinate) is
\[ \frac{1}{2} \log_2 \left( \frac{v+1}{2} \right) - 1 \sum_{d'=0}^{d} 2^\frac{1}{2} \left( \log_2 \left( \frac{v+1}{2} \right) - 2d' - 2 \right) = \left( \frac{v+1}{2} \right)^\frac{1}{2} - 1. \]

Therefore, since each processor occupies the square of unit side length centred on its location, the side length of the bounding square is
\[ 1 + 2 \left( \left( \frac{v+1}{2} \right)^\frac{1}{2} - 1 \right) \]
and the area of the bounding square is
\[ 4 \left( \frac{v+1}{2} \right) - 4 \left( \frac{v+1}{2} \right)^\frac{1}{2} + 1 = \Theta(v). \]

The proof is extended to trees of odd depth \( d_{odd} \) by the observation that the minimal bounding rectangle of such a tree is sandwiched between those of the trees of even depth \( d_{odd} - 1 \) and \( d_{odd} + 1 \) which have approximately half and twice as many vertices respectively. This is sufficient to satisfy the asymptotic result.

### 3.2.3 A More General Technique – \( k = 4 \) and Beyond

As noted in section 3.1 natural algorithms for many problems might be better described in terms of a \( k = 4 \) (or more) RDC skeleton. Again, the obvious idealised implementation would be on a complete tree of processors in which each non-leaf has \( k \) children. Clearly, with each processor requiring degree \( \geq 5 \), no direct counterpart of the H-tree exists. However, the layout in figure 3–2 shows that the \( k = 4 \) tree can be embedded into the grid in linear area. Furthermore, it will become clear in section 3.3 that the execution time of the new skeleton is increased by only a constant factor from some imaginary degree 5 counterpart of the H-tree.
All processors are marked with a \( \cdot \) and are active as leaves. Processors also active at higher levels are marked with increasing numbers of circles. The root is marked with a solid circle and is active at all levels. Thus, there are four processors active at the level below the root, sixteen at the next level down and so on, as required by the 4-ary tree.

Figure 3-2: The 4-ary tree layout.
Chapter 3. The Recursive Divide & Conquer Skeleton

The layout exploits the fact that levels of the idealised processor tree are active in strict sequence and independently (except during transfer from one level to the next). Thus each grid processor responsible for the operations of a tree vertex at some level is also made responsible for one of the children of that vertex at the next level down. Having divided a problem in four, each grid processor keeps one sub-problem and dispatches the other three. The structure of the layout is such that two of the three receive their problems along the direct path from the processor which was the parent (and is now a sibling), while the third receives its problem in two halves, via the other two children. Figure 3–3 illustrates this point for transfers in the upper left-hand quadrant, labelling the sub-problems \( p_1, p_2, p_3, p_4 \) with \( p_1 \) being the sub-problem which does not move. Note that the structure of the layout ensures that the processors which forward information along the paths from \( p_1 \) to \( p_2, p_3 \) and \( p_4 \) are not active in any other capacity when required to do so. The situation in the other quadrants is illustrated by an appropriate rotation of figure 3–3. Dispatching the two halves of \( p_4 \) first allows the second halves of their journeys to be overlapped in time with the distribution of problems \( p_2 \) and \( p_3 \).
Chapter 3. The Recursive Divide & Conquer Skeleton

The regularity of the technique allows it to be used between each successive pair of virtual levels, and in reverse with the results. It also ensures that each processor knows independently, at all times, whether it should be active in the computation, simply forwarding information or inactive, as a simple function of its absolute location in the grid.

3.2.4 The $k$-way Recursive Split

It is interesting to ask whether the technique described above can be efficiently generalised to deal with a $k$-way split, for an arbitrary constant $k$. The original method has two important properties. Firstly, its regularity allows easy theoretical analysis and easy practical implementation. Secondly, the portion of time delay introduced simply by the distance covered in data transfer from root to leaves is asymptotically optimal (at $\Theta(\sqrt{n})$).

A loose description of the generalised layout technique emerging from it is as follows:

- Suppose that the two closest factors of $k$ are $l$ and $m$ (i.e. $l \times m = k$ and $|l - m|$ is minimal for any such integers $a, b$ with $a \times b = k$). Then the grid used must be of dimension $l^{\log_k n} \times m^{\log_k n}$. This is notionally split into $k$ sub-grids of dimension $l^{(\log_k n)-1} \times m^{(\log_k n)-1}$, each of which will deal with one of the $k$ sub-trees of the root. This splitting is repeated recursively until the grid has been divided into $k^{(\log_k n)-1}$ groups of $l \times m$ grid vertices which can now be mapped to the $k$ vertices of the appropriate sub-tree. In each group, the vertex nearest to the centre of the grid is nominated as the “parent”. This is the vertex which is also active at the tree level above the leaves and which will distribute problems to the other leaves in its cluster. Parents at higher levels are then recursively selected by the same rule from the $k$
parents at the next level down in each sub-tree, until we have one vertex nominated as the root. This will be the vertex at the centre of the grid if both \( l \) and \( m \) are even, or one of the two or four vertices equidistant from the centre otherwise. At any level, the choice between such equidistant vertices is arbitrary but must be consistent between levels to preserve regularity.

The following theorem notes that layouts generated by this technique cannot have optimal route to leaf distance if \( k \) is not a perfect square.

**Theorem 2** If \( k \) is not a perfect square then the layout of a \( k \)-way split tree with \( n \) leaves produced by the technique above has at least one leaf to which the optimal physical path from the root is of length (and therefore time delay) \( \not= O(\sqrt{n}) \).

**Proof**: Let \( l \) and \( m \) be the two nearest factors of \( k \), with \( l : m = r : 1, r > 1 \). The tree is of depth \( \log_k n \) and let the bounding rectangle of the whole layout be of dimension \( s \times t \). Supposing without loss of generality that \( s \geq t \), then

\[
\begin{align*}
\text{We know } st & \geq n \text{ and so } s \geq \left( r^\log_k n \right)^{1/2}. \\
\text{Noting that } \log_k n & = \frac{\log_k n}{\log_k k}, \text{ define } \frac{1}{\log_k k} = k', \text{ with } k' \geq 0 \text{ since } k \geq r. \\
\text{Thus, } s & \geq \left( r^{k' \log_k n} n \right)^{1/2} = \left( n^{1+k'} \right)^{1/2} \neq O(\sqrt{n}), \\
\text{and so } s & \not= O(\sqrt{n}).
\end{align*}
\]

Wherever the root is mapped, there must be some leaf whose optimal path to the root is of length at least \( \frac{s}{2} \neq O(\sqrt{n}) \), since every processor takes an active part as a leaf. •

Thus, to use the full square grid with this technique, \( k \) must be selected to be a perfect square.
3.3 Analysis of Implementations

In this section we consider the performance which can be expected of the proposed implementations. The abstract specification presented in section 3.1 allows complete flexibility in the definition of the "split$_k$" function. This introduces the possibility that certain instances may be decomposed in a way which leaves large portions of the idealised processor tree unused. However, both the sorting example of section 3.1 and the further examples of section 3.5 share an important property – the instances generated by a particular call of "split$_k$" are of essentially the same size, allowing for minor discrepancies (e.g. the two "halves" of a list of odd length). The analysis presented applies to any example which shares this property. The desirability of well balanced sub-division is familiar from study of sequential algorithms [2].

In this context, the notion of size may have two connotations. It describes both the quantity of data required to represent each sub-instance and the amount of further divisibility inherent in each. It is important to note that these are not necessarily identical. For example, an instance of the problem of finding the maximum value of some function over a domain of equally separated points can be described by two items of a data (the first point and the separation) but can be recursively divided into $n$ trivial sub-instances. To avoid confusion, the term "grain" is introduced to denote size in the latter sense. Thus, an instance of grain $n$ can be recursively divided into $n$ base-case instances, conceptually evaluated at the leaves of the complete $k$-ary tree of processors.

The detailed analysis presented below considers the implementation of such instances of grain $n$, with $k = 2$, on a grid embedded H-tree of $2n - 1$ processors. Performance is compared with that achievable by a single processor solving the
same instance. One assumption made, that the size of the grid will always be
large enough to accommodate the instance directly, is unrealistic. In section 3.4
the implications of a fixed size grid are considered and analysed. An associated
technique, of not fully expanding the idealised tree on the grid, is shown to widen
the range of examples for which optimal efficiency (with respect to a single pro-
cessor) is obtainable. Finally, the existence of similar results for the generalised
implementation technique of section 3.2.3 is noted.

3.3.1 The Full Binary Tree

We now consider the performance of the H-tree implementation of the binary
RDC skeleton. To analyse any particular problem there are six parameters which
must be considered. These are

\( s(n) \): the time to split a problem instance of grain \( n \) into two sub-problems of
grain \( \frac{n}{2} \),

\( p(n) \): the number of constant size packets of information required to describe a
problem instance of grain \( n \),

\( r(n) \): as \( p(n) \), for the result of a problem of grain \( n \),

\( j(n) \): the time to join two sub-results of grain \( \frac{n}{2} \) to form one super-result of grain
\( n \),

\( f(1) \): the time taken to evaluate the base case function on an indivisible instance
(i.e. of grain 1),
$d(i)$: the number of links on the grid layout between a tree vertex at $i$ stages from the leaves and its sons, this being

$$d(i) = \begin{cases} 
2^\frac{i}{2}(i-1) & \text{for odd } i, \\
2^\frac{i}{2}(i-2) & \text{for even } i.
\end{cases}$$

The assumed model for individual processors allows data to be transferred simultaneously on all links. Thus, since the examples under consideration generate sub-instances of equal size, similar activities will occur concurrently across successive levels of the virtual binary tree. The generality and regularity of the layout mean that the total execution time for a problem of grain $n$ on a tree with $n$ leaves is asymptotically the same as the time to execute all the instructions on any path from root to leaf. This has three phases:

1. sub-division and distribution of the problem, taking time

$$\sum_{i=1}^{\log_2 n} \left[ s(2^i) + p(2^{i-1}) + (d(i) - 1) \right]$$

where the last term introduces the delay encountered due to path length between successive levels,

2. computation of the base case at the leaves, taking time denoted by $f(1)$,

3. return and combination of results (including path delays), taking time

$$\sum_{i=1}^{\log_2 n} \left[ j(2^i) + r(2^{i-1}) + (d(i) - 1) \right].$$

The execution time of this parallel implementation of the skeleton, denoted $T_{P2}(n)$ is simply the sum of these phases. The $d$'s are independent of the algorithm and can be removed from the summation and simplified, giving

$$T_{P2}(n) = 4(\sqrt{n} - 1) + f(1) - 2\log_2 n$$

$$+ \sum_{i=1}^{\log_2 n} \left[ s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1}) \right]$$
for cases in which the depth $\log_2 n$ is even. When the depth is odd, the first term is replaced by $3\sqrt{2n} - 4$.

### 3.3.2 Optimal Efficiency

We now consider conditions under which the $\Theta(n)$ processor H-tree implementation evaluates instances of grain $n$ with optimal efficiency. This will be judged with respect to the performance obtainable using a single processor, by comparing the resources used in each case, namely the product of the number of processors and the execution time.

The following lemmas, applying to the variables as previously described, are required in the subsequent analysis. The function $h(n)$ may be replaced by any of $s(n), j(n), p(n)$ and $r(n)$. All are proved by simple manipulations.

**Lemma 1**

\[
\sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s(2^i) + j(2^i) \right] = \Omega \left( \sum_{i=1}^{\log_2 n} \left[ s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1}) \right] \right)
\]

$\iff s(n) = j(n) = p(n) = r(n) = 0$.

**Lemma 2**

\[h(n) = O(n^a), a > 0, \Rightarrow \sum_{i=1}^{\log_2 n} h(2^i) = O(n^a)\]

**Lemma 3**

\[h(n) = O(n^a) \Rightarrow \sum_{i=1}^{\log_2 n} \frac{1}{2^i} h(2^i) = \begin{cases} O(n^{a-1}) & \text{if } a > 1, \\ O(\log n) & \text{if } a = 1, \\ O(1) & \text{if } a < 1. \end{cases}\]

A straightforward sequential evaluation would (ignoring constant overheads dealing with recursion) perform all the "real" operations of each vertex in the
Chapter 3. The Recursive Divide & Conquer Skeleton

processor tree. This excludes those associated purely with data transfer, which are redundant. Thus, the time taken by such an implementation would be

\[ T_{S_2}(n) = n f(1) + n \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s\left(2^i\right) + j\left(2^i\right)\right]. \]

The asymptotic efficiency of the grid tree with respect to this sequential evaluation is now investigated.

The tree certainly cannot be more efficient, since processors are often idle and the algorithm is the same, but it is interesting to investigate conditions which allow it to be asymptotically of equal efficiency, i.e. for which the resources used are asymptotically identical,

\[ 1 \cdot T_{S_2}(n) = \Theta(n T_{P_2}(n)). \]

In such a situation the skeleton will be said to be implemented with optimal efficiency. Obviously such an analysis will also serve to highlight those properties which algorithms should exhibit if they are to allow good though not necessarily optimal implementation.

Since \( T_{S_2}(n) = O(n T_{P_2}(n)) \) (it executes the same operations without communication overheads) it is only necessary to investigate

\[ T_{S_2}(n) = \Omega(n T_{P_2}(n)) \]

i.e.

\[ n f(1) + n \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s\left(2^i\right) + j\left(2^i\right)\right] \]

\[ = \Omega\left(n^{\frac{3}{2}} + n f(1) + n \sum_{i=1}^{\log_2 n} \left[s\left(2^i\right) + j\left(2^i\right) + p\left(2^{i-1}\right) + r\left(2^{i-1}\right)\right]\right) \]

The \( n f(1) \) term on the right can be removed, as (by lemma 1) can the \( n \sum_{i=1}^{\log_2 n} \left[ s\left(2^i\right) + j\left(2^i\right) \right] \) term (while noting that optimal efficiency is achieved, somewhat obscurely, in the
Chapter 3. The Recursive Divide & Conquer Skeleton

case \( s(n) = j(n) = p(n) = r(n) = 0 \), when nothing happens at all. This leaves
\[ nf(1) = \Omega \left( n^{\frac{3}{2}} + n \sum_{i=1}^{\log_2 n} \left[ s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1}) \right] \right) \]
with \( s, j, p, r \) not all zero, as a requirement for optimal efficiency. This will hold only when

- \( f(1) = \Omega (\sqrt{n}) \), and

- \( f(1) = \Omega \left( \sum_{i=1}^{\log_2 n} m(2^i) \right) \), in cases when \( s, j, p, r \) are all \( O(m(n)) \). Note that if \( m(n) = O(n^a) \), \( a > 0 \) then \( f(1) = \Omega (n^a) \) is sufficient by lemma 2.

This means that the goal of optimally efficient implementation of the skeleton is only achievable for algorithms in which the work done in the base case is asymptotically as large as both \( \sqrt{n} \), and the largest of the quantities represented by all the occurrences of \( s, j, p \) and \( r \) in the sequence of evaluation which lead directly to the call of the base case. In terms of the tree, this means that the work done at each of the leaves (which account for half the available processing power) must be asymptotically as large as all the work done getting there, including communication time. This in turn is guaranteed to be \( \Omega (\sqrt{n}) \) since the leaves are physically this number of links from the root.

It is not difficult to see that analysis of the \( k = 4 \) skeleton follows a similar pattern to the two way split, with an additional slow-down introduced by the sharing of edges giving
\[ T_{P4}(n) = f(1) + \sum_{i=1}^{\log_4 n} \left[ s(4^i) + j(4^i) + \frac{3}{2} \left( p(4^{i-1}) + r(4^{i-1}) \right) \right] + 4(d(i) - 1) \]
where \( f(1), s, j \) and \( p \) have the usual meanings and \( d(i) \) is the path length between a processor dealing with a super-problem associated with a vertex at \( i \) stages from the leaves and its two nearest children. Noting from the layout presented in figure 3–2 that \( d(i) = 2^{i-1} \) for \( 1 \leq i \leq \log_4 n - 1 \) and \( d(\ log_4 n) = 1 \) the
Chapter 3. The Recursive Divide & Conquer Skeleton

$d$'s are simplified to obtain

\[ T(n) = f(1) + 2\sqrt{n} - 4 \log_4 n + \sum_{i=1}^{\log_4 n} \left[ s(4^i) + j(4^i) + \frac{3}{2} \left( p(4^{i-1}) + r(4^{i-1}) \right) \right] \]

By analogy with the binary tree it can be seen that

\[ T_{S4}(n) = nf(1) + n \sum_{i=1}^{\log_4 n} \left[ s(4^i) + j(4^i) \right]. \]

It is a simple exercise to show that similar results to lemmas 1-3 and their consequences hold. Thus the four-way recursive split provides an alternative skeleton to the two-way recursive split, with similar properties.

Finally, consider the technique of section 3.2.3 for general $k$. Theorem 2 illustrates the problems involved when $k$ is not a perfect square, owing to the longer than $O(\sqrt{n})$ root to leaf path length. However, in cases where $k$ is a perfect square, this objection does not apply, since the whole grid and all the sub-tree building blocks will be square giving all shortest paths $O(\sqrt{n})$.

The only remaining problem in repeating the above results is to ensure that the distribution of $k - 1$ sub-problems by parent nodes at each level does not introduce an asymptotically significant slow-down. Using notation similar to that dealing with the H-tree layout, let $d(i)$ denote the length of the shortest path between a parent at $i$ stages from the leaves and its most distant child. Note that the grid is square and that paths do not double back on themselves. Therefore, the total contribution of edge delays on any direct root to leaf path is

\[ \sum_{i=1}^{\log_4 n} (d(i) - 1) = O(\sqrt{n}). \]

Furthermore, the observation that the layout assigns distinct sub-trees at every level to non-overlapping areas of the grid makes it clear that there is no interference between messages transferred inside such distinct subtrees. It is now clear that even the simplest procedure, in which a parent deals with its $k$ children sequentially, by the most direct paths, is good enough to meet the bound. In this
situation the total time lost to path-length delays will be

\[ \sum_{i=1}^{\log_k n} (k - 1) (d(i) - 1) = (k - 1) \sum_{i=1}^{\log_k n} (d(i) - 1) = O(\sqrt{n}), \]

remembering that \( k \) is constant with respect to \( n \) (which in practice means assuming that \( n \) is sufficiently large to make \( k \) insignificant). The example of the 4-way split shows how communications may be overlapped in some regular manner to improve upon the constants involved. However, the asymptotic result provided by this simplest method is good enough to ensure that similar results to those above extend to any perfect square \( k \).

### 3.4 Partial Trees

The drawback of implementation on a complete binary tree with depth \( \log_2 n \) was that algorithms had to be heavily biased towards giving the leaves large amounts of work in order to achieve optimal efficiency. Unfortunately, it seems probable that in many algorithms designed with the recursive split skeleton the time taken to evaluate the base case at the leaves will be too small to outweigh the time spent getting to them. Furthermore, it is unreasonable to assume that any fixed size machine will be large enough to handle, in the straightforward way, any instance presented. For these reasons it seems worthwhile to consider the execution of instances of arbitrary grain on fixed size machines. This is achieved by simulating the operations of the lower (and now physically non-existent) levels at higher levels. In this way, instances of grain \( n \) will be evaluated on a binary tree of depth \( < \log_2 n \). Evaluation will proceed as before from the root to the physical leaves. These will receive problems which have not yet reached the base case. They simulate the actions of all their virtual descendants until they have solved their sub-problems. These results are then passed back up the physical tree
Chapter 3. The Recursive Divide & Conquer Skeleton

and evaluation is completed as before. In terms of the preceding analysis there are two significant changes. Firstly, the time taken to evaluate a problem will be increased by the simulation (although the data transfers between the simulated levels are free). Secondly, the number of processors used (i.e. the size of the physical tree) and hence the grid area will be reduced. The trade-off between these factors is now investigated.

In such trees, leaves deal with problems of grain \(2^c\), \((1 \leq c \leq \log_2 n - 1)\), by simulating the actions of their imaginary descendants. There will be \(\frac{n}{2^c}\) leaves and thus \(\frac{n}{2^{c-1}} - 1\) vertices, and the tree will be of depth \(\log_2 \left(\frac{n}{2^c}\right)\). For the H-tree implementation the execution time will be

\[
T_{P2}^c(n) = \sum_{i = c+1}^{\log_2 n} [s(2^i) + j(2^i) + p(2^{i-1}) + r(2^{i-1})] \\
+ 2 \log_2 \left(\frac{n}{2^c}\right) [d(i) - 1] + 2^c \left(f(1) + \sum_{i=1}^{c} \frac{1}{2^i} [s(2^i) + j(2^i)]\right)
\]

where the first term represents the normal part of the tree execution and the third term represents the simulated phase. The asymptotic efficiency of such trees with varying \(c\) is now considered.

Simplifying the term in \(d\), multiplying by the size of grid used and discarding the lower order terms produced, shows that the product of processors used and time taken is

\[
= \Theta \left( \frac{n}{2^c} \sum_{i = c+1}^{\log_2 n} [s, j, p, r] + \left(\frac{n}{2^c}\right)^{\frac{3}{2}} + nf(1) + n \sum_{i=1}^{c} \frac{1}{2^i} [s, j] \right).
\]

By the same arguments advanced for the full tree, no partial tree can ever outperform the sequential evaluation by this measure of efficiency. Thus, only conditions which allow

\[
T_{S2}(n) = \Omega \left( \frac{n}{2^c} T_{P2}^c(n) \right)
\]
Chapter 3. The Recursive Divide & Conquer Skeleton

need be considered. Removal of redundant terms leaves

\[ nf \left( 1 \right) + n \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s \left( 2^i \right) + j \left( 2^i \right) \right] = \]

\[ \Omega \left( \left( \frac{n}{2^c} \right)^{\frac{3}{2}} + \frac{n}{2^c} \sum_{i=c+1}^{\log_2 n} \left[ s \left( 2^i \right) + j \left( 2^i \right) + p \left( 2^{i-1} \right) + r \left( 2^{i-1} \right) \right] \right). \]

The simulation of only a constant number of the lowest levels can have no effect on the asymptotic situation. Therefore the effect of simulating some constant fraction of the levels is considered, taking \( c = C \log_2 n \) with \( 0 < C < 1 \) indicating that the bottom fraction \( C \) of the idealised tree levels are simulated by the leaves of the physical tree.

With \( c = C \log_2 n \), the partial tree is optimally efficient when

\[ nf \left( 1 \right) + n \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s, j \right] = \Omega \left( \left( n^{1-C} \right)^{\frac{3}{2}} + n^{1-C} \sum_{i=C \log_2 n + 1}^{\log_2 n} \left[ s, j, p, r \right] \right) \]

and thus when,

\[ f \left( 1 \right) + \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s, j \right] = \Omega \left( \left( n^{\frac{1}{2}-C} \right)^{\frac{3}{2}} + n^{1-C} \sum_{i=C \log_2 n + 1}^{\log_2 n} \left[ s, j, p, r \right] \right). \]

It appears that \( C = \frac{1}{3} \) is an important value and the cases on either side of it are now analysed.

Suppose \( C < \frac{1}{3} \), and that \( s, j, p, \) and \( r \) are all \( O \left( n^a \right) \) for some \( a > 0 \). If \( a \leq C \) then analysis similar to the full tree case shows that

\[ f \left( 1 \right) = \Omega \left( \left( n^{\frac{1}{2}-C} \right)^{\frac{3}{2}} \right) \]

gives optimal efficiency. With \( a > C \)

\[ f \left( 1 \right) = \Omega \left( \left( n^{\frac{1}{2}-C} \right)^{\frac{3}{2}} + n^{a-C} \right) \]

is required. More interestingly, suppose that \( C \geq \frac{1}{3} \), and that \( s, j, p \) and \( r \) are all \( O \left( n^a \right) \). Then for optimal efficiency it is sufficient that

\[ f \left( 1 \right) + \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[ s, j \right] = \Omega \left( n^{a-C} \right) \]
Chapter 3. The Recursive Divide & Conquer Skeleton

If \( a > C \) then (by lemma 3) the second term is never sufficient to outweigh \( n^{a-C} \) and \( f(1) = \Omega(n^{a-C}) \) is required for optimal efficiency. However, if \( a \leq C \) all that is required is

\[
f(1) + \sum_{i=1}^{\log_2 n} \frac{1}{2^i} \left[s \left(2^i\right) + j \left(2^i\right)\right] = \Omega(1),
\]

which is true provided that \( f(1) \neq 0 \), i.e. that the idealised leaves actually do something. The important point is that the work required of the idealised leaves to ensure optimal efficiency is no longer dependent upon \( n \).

An example which takes advantage of this result is presented in the next section.

3.5 Examples

3.5.1 The Discrete Fourier Transform

An algorithm to compute the DFT of \( n \) points can be described very neatly with the simple \( k = 2 \) RDC skeleton [58]. The crucial observation is that the DFT of an \( n \) point vector may be computed as a simple arithmetic combination of the DFTs of two vectors of \( \frac{n}{2} \) points, these instances being derived directly from the original vector.

For analysis, the important characteristics of the algorithm are that

\( p(n) = n \), the \( n \) points to be transformed,

\( r(n) = n \), the \( n \) transformed points,

\( s(n) = 0 \), since no computation is required to split the vector of points into two vectors (although points must be dispatched in shuffled order), and
Chapter 3. The Recursive Divide & Conquer Skeleton

\[ j(n) = \frac{3n}{2}, \] consisting of \( \frac{n}{2} \) multiplications and \( n \) additions.

\[ f(1) = 1, \] the leaf operation being a single multiplication,

Thus,

\[
T(n) = 4(\sqrt{n} - 1) + 1 - 2\log_2 n + \sum_{i=1}^{\log_2 n} \left[ 3\left(2^i\right) + 2^{i-1} + 2^{i-1} \right]
= 5n + 4\sqrt{n} - 2\log_2 n - 8.
\]

A useful method of evaluating the relative efficiency of parallel algorithms is to compare the product of the time and number of processors involved. Such comparisons have the re-assuring property that an optimal \( n \) processor algorithm uses the same resources (by this measure) as an optimal one processor algorithm for the same problem. With the H-tree layout the RDC skeleton algorithm uses \( \Theta(n) \) processors and thus uses \( \Theta(n^2) \) resources. In terms of efficiency this doesn’t compare favourably with a straightforward single processor evaluation which can be executed in time \( \Theta(n \log n) \), thus using \( \Theta(n \log n) \) resources. The DFT of \( n \) points may also be computed in \( \Theta(\log n) \) time on \( \frac{n}{2} \) processors connected into a 2-shuffle network (with a simple adaption of the well known algorithm in [63]). However, [10] indicates that any grid embedding of this network without overlapping paths requires \( \Omega\left(\frac{n}{\log n}\right)^2 \) processors in our model and must have at least one edge traversing \( \Omega\left(\frac{n}{\log^2 n}\right) \) processors. Since each edge of the shuffle graph is used at each step of the algorithm any such optimally embedded implementation would take time \( \Theta\left(\frac{n}{\log n}\right) \) thus using \( \Theta\left(\frac{n}{\log n}\right)^3 \) resources, which is inferior to the RDC skeleton method. This surprising result is produced by the weakness of any such shuffle implementation rather than the shuffle algorithm itself.

A different result can be obtained by considering the shuffle DFT as an algorithm for an EREW SM machine, with each shuffle step being simulated by two EW operations. Each step of such a machine can be simulated on the grid of
\( \Theta(n) \) processors in \( \Theta(\sqrt{n}) \) time (e.g. by using the well known sorting method of \([64]\)). The whole algorithm can therefore be executed in time \( \Theta(\sqrt{n \log n}) \), using \( \Theta(n^{3/2} \log n) \) resources, which is superior to both other methods. However, the RDC solution is certainly the most easily specified. It should also be noted that the RDC skeleton assumes that the input is initially located at a single processor, thus incurring substantial distribution costs. All the other solutions assume the input to be already distributed. Whether or not this is significant depends upon the number and location of connections to the "outside world".

### 3.5.2 Approximate Integration

The approximate integration of some continuous function \( y(x) \) over an interval \([a, b]\) is an obvious candidate for evaluation using the RDC skeleton, given the observation that

\[
\int_a^b y(x) \, dx = \int_a^{b/2} y(x) \, dx + \int_{b/2}^b y(x) \, dx.
\]

In order to use the results obtained previously, it is necessary to consider what is meant by the grain of such a problem. For any particular instance, this will depend upon the interval width \( \delta \) at or below which an approximation to the integral will suffice (thus requiring no further sub-division) i.e.

\[
\int_c^{c+\delta} y(x) \, dx = \text{approx}(y, c, \delta).
\]

Given a function \( y \) and its minimum interval \( \delta \), an instance of the integration problem over interval \([a, b]\) will be of grain \( n = 2^\left\lfloor \log_2 \frac{b-a}{\delta} \right\rfloor \).

Consider the use of Simpson's rule for the approximate evaluation of the base-case integrals. This requires the evaluation of \( y \) at \( w \) (an arbitrary odd constant) equally spaced points including the end-points for each minimum interval, followed by a simple arithmetic combination of these values involving \( \Theta(w) \) operations. Using the terminology defined previously, it can be seen that
Chapter 3. The Recursive Divide & Conquer Skeleton

- \( s(n) = 2 \), the cost of calculating \( \frac{k-a}{2} \),
- \( j(n) = 1 \), a single addition,
- \( p(n) = 2 \), the two endpoints,
- \( r(n) = 1 \), the approximated result, and
- \( f(1) = \Theta(w + wY) \), where \( Y \) is the cost of each evaluation of \( y \).

The results of section 3.3 show that, for the evaluation of a grain \( n \) integral on a full grid-embedded tree with \( n \) leaves, optimal efficiency is only possible when \( w + wY = \Omega(\sqrt{n}) \). This is not very satisfactory, since it requires that the number of points used in the Simpson approximation or worse, the complexity of evaluating \( y \), should depend upon the grain of the problem instance. However, the analysis for partial trees is more encouraging. Since all of \( s, j, p \) and \( r \) are \( O(n^{\frac{1}{2}}) \), any tree with \( C \geq \frac{1}{3} \) requires that \( f(1) \neq 0 \) for optimal efficiency. This holds with any \( w \neq 0 \) i.e. for any use of Simpson’s rule irrespective of the number of evaluation points used.

3.5.3 Matrix Multiplication

The final example, matrix multiplication, uses the generalised RDC skeleton. The algorithm presented here is adapted from [30].

Consider the problem of multiplying two square, \( m \) element matrices \( A \) and \( B \) to produce a third, \( C \). By partitioning each matrix into four quadrants, we have

\[
\begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\cdot
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
=
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
\]

and it can be seen that \( C_{ij} = A_{i1} \cdot B_{1j} + A_{i2} \cdot B_{2j} \), for \( i \) and \( j \in \{1, 2\} \). Thus, the problem of computing \( C \) can be described in terms of eight \( \frac{m^2}{4} \) element matrix
multiplications and four $\frac{m}{4}$ element matrix additions. This is just an RDC skeleton algorithm with $k = 8$. The base case occurs when $m = 1$ and is a simple scalar multiplication.

It is important to note that an instance of grain $n$, by the definition of section 3.3, does not involve $n$ element matrices. Consider an instance which does decompose into $n$ base case instances (and which is therefore of grain $n$). At each combination of sub-results, the solutions to eight sub-instances are combined to solve a single instance involving a matrix with four times as many elements. In this way, $n$ base case results eventually combine to produce a final matrix with $n^{\log_4 4}$ elements. In terms of the measures of section 3.3 this gives, $p(n) = 2n^{\log_4 4}$, the two input matrices, $r(n) = n^{\log_4 4}$, $s(n) = 0$, $j(n) = n^{\log_4 4}$ and $f(1) = 1$.

In practice, the position is complicated by the observation in section 3.2.3 that in order to use the square grid the solution must be specified with $k = 9$, a perfect square. The ninth sub-instance will just be an empty dummy, but will still use resources.

Thus, we consider instances involving $m$ element matrices (which are therefore of grain $n = m^{\log_4 9}$ with $k = 9$). Using the simplest problem distribution method, by which a processor at one level distributes sub-instances sequentially, the RDC skeleton implementation will execute in time

$$T_{P_9}(m) = 2 \left[ 9 \left( \frac{m}{4} \right) + 9 \left( \frac{m}{4^2} \right) + \ldots + 9 \right] + m^{\frac{1}{2} \log_4 9} + 1 + 4 + 16 + \ldots + \frac{m}{16} + \frac{m}{4} + m = O(m).$$

The terms preceded by 9's represent the transfer of information, the $m^{\frac{1}{2} \log_4 9}$ is the delay across the $m^{\log_4 9}$ processor grid, the 1 is the base case calculation and the other terms represent the additions involved in combining sub-results.
Chapter 3. The Recursive Divide & Conquer Skeleton

The dummy instance would be unnecessary for a sequential implementation. Here, the time taken for instances involving size $m$ matrices, $T_{SS}(m)$, is described by the recurrence relation

$$ T_{SS}(m) = 8T_{SS}(\frac{m}{4}) + m, $$
$$ T_{SS}(1) = 1. $$

which has solution $T_{SS}(m) = O\left(m^{\frac{3}{2}}\right)$, the same as the standard sequential matrix multiplication algorithm. Thus, the $k = 9$ RDC skeleton implementation offers a significant, although sub-optimal, improvement in performance.
Chapter 4

The Task Queue Skeleton

The skeleton presented in this chapter is of more general applicability than that of chapter 3. In fact it would be possible, though inadvisable for reasons of efficiency, to implement the RDC skeleton in terms of the task queue. The task queue skeleton differs from the other three discussed in the thesis by having an explicitly parallel abstract specification. Section 4.1 motivates and introduces this specification. Section 4.2 introduces the overall structure of a proposed implementation which is discussed in detail in sections 4.3 and 4.4. The implementation is summarised in section 4.5. Finally, section 4.6 presents some examples for which solutions may be described in terms of the task queue skeleton.

4.1 The Abstract Specification

4.1.1 Motivation

The task queue skeleton aims to encompass a somewhat more general group of algorithms than does the recursive divide and conquer skeleton of chapter 3. Task queue algorithms are explicitly parallel and share two properties. Firstly, they all
deal with problems for which both instances and solutions may be represented in terms of some large, shared data structure. Secondly, they all proceed from problem to solution by repeated concurrent execution of a procedure (or "task") which manipulates some part of the data structure with respect to certain others. Execution of each such task results in the contents of the data structure progressing towards a description of a solution to the problem. In executing some instance of the task, the algorithm may generate details of further task instances. These are added to a pool of such instances, the "task queue", according to some selected queuing discipline. The flavour of the skeleton is probably best illustrated by a particular example.

The one-to-all shortest path problem for an arbitrary weighted graph involves finding the shortest paths (in terms of weight of edges traversed) from one specified "central" vertex to each of the others. Deo et al. [20] present a parallel algorithm (based on [50]) which solves this problem for graphs containing no negative cycles. (In such cases the problem has no solution and the algorithm fails to terminate). In a task queue solution, the graph is represented by an array (situated in the shared data structure) of records indexed by vertex identifiers, one per vertex, with a field to record the shortest known path to the central vertex, and another field giving details of the connectivity of the indexed vertex (eg. a boolean array or a linked list). Each shortest path field is initialized to \( \infty \) except that of the chosen vertex which is initialized to zero. The task procedure is parameterised by a vertex identifier. This denotes a vertex to which an improved (i.e. shorter) path has been found. The procedure looks at all the neighbours of the vertex to see if new shortest paths to them are implied by this discovery, by checking edge weights. The shortest path fields of any such vertices are updated and new tasks instances are created by adding a copy of the identifier of the updated vertex to the queue of parameters. The task queue is initialized to contain a single task parameter, the identifier of the selected central node, to which a "shortest path"
REPEAT
    IF tasks are available THEN try to grab a task;
    IF successful THEN BEGIN
        execute the task;
        place any tasks created on the queue;
    END;
UNTIL no tasks available AND all processors are inactive;

Figure 4–1: Idealised processor code

of length zero has been “discovered”. The ordering on the queue is not important
for correctness, but a Last-In-First-Out discipline may tend to produce results
faster, since it immediately follows up the newest paths found.

The obvious source of parallelism in the execution of the skeleton underpinning
such algorithms is the concurrent execution of large numbers of task instances.
Processors can repeatedly grab parameters from the task queue and execute cor-
responding instantiations of the task procedure.

4.1.2 Specification

A more precise abstract specification of the task queue skeleton is now desirable.
The user must consider the skeleton to be executed on an unspecified number of
parallel processors, each independently executing the loop shown in figure 4–1.
All processors have read and write access to the task queue and to the shared
data structure describing the problem instance. Additionally, the processors may
perform simple indivisible arithmetic operations on the data structure. For exam-
ple, the operation “add c to location x” is performed as a single operation, rather
than as a fetch followed by add and write back. Clashes between such operations will be discussed shortly.

At any point of a computation an abstract processor will be involved in executing one of four types of instruction:

- adding a task to the queue,
- removing a task from the queue,
- accessing the data structure,
- a "local" instruction involving neither queue nor data.

The user may assume that each instruction takes the same time and therefore that processors proceed synchronously. Thus at each time step, $p \leq n$ attempts are made to place tasks on the queue and $r \leq n$ attempts made to remove tasks from the queue. The machine of the abstract specification is capable of performing these operations with no overhead. In a single time step, the $p$ new tasks are placed on the queue and the requests are served with the $r$ most appropriate tasks available (according to the queuing discipline). These may include some of the $p$ tasks just placed on the queue.

It is important to note that the user may make no assumptions about the number of processors executing the skeleton. This introduces an element of non-determinism into the overall pattern of abstract execution. Depending upon the number of processors actually involved, certain tasks may or may not be activated simultaneously. Consequently, the task instances which they generate may be added to the queue in different positions, according to circumstances (e.g. consider using a first-in first-out discipline). This in turn affects the order in which tasks are subsequently executed, and so on. It also has implications for the shared
data structure. Since the user cannot, in general, be certain about the number of
tasks executed simultaneously, no assumptions can be made about concurrency
between instructions within tasks. For different numbers of processors, a certain
instruction from one task may or may not be executed before some other instruc-
tion from another task. The only guarantee is that all instructions from one task
will be executed before any from its direct descendants (since the addition of new
descriptors to the task queue is the last operation of a task). Thus, a user can
make no assumptions (beyond the above) about the ordering of two independent
accesses to the same shared location. The concept of clashing accesses therefore
has no meaning at the user's abstract level, since its occurrence can never be
determined without knowledge of the implementation. It is simply guaranteed
that all accesses will be made eventually, in an order subject only to the previous
guarantee. Although this initially appears awkward, the examples of section 4.6
suggest that this poses no difficulties for a wide range of problems.

In summary, to describe a task queue solution to some problem, the user must
provide four items:

- a “type” specification of the data structure to be used to describe the evolv-
ing solution,

- a “variable” of this data structure type with contents describing the initial
problem instance,

- a parameterized “task” procedure describing the operations by which some
part of the data structure may be adjusted to lead towards the solution, where
the parameter specifies the part of the data structure upon which operations are centered. As a result of the operations performed, the proce-
dure may (as its final action) choose to create arbitrarily many new instances
of the task to be executed later, by adding details of their parameters to
the task queue. The procedure may also reserve local variables. These are not accessible by any other task instantiation.

- an initial collection of parameters describing specific instances of the generic task, together with a queuing discipline (selected from some set of options) controlling maintenance of the task queue.

Subsequent sections in this chapter discuss techniques which could be used to implement the abstract "task queue machine" on the grid.

4.2 The Structure of an Implementation

The abstract specification of the skeleton was interwoven with a description of an idealised parallel machine which could execute it. The processors of the idealised machine were capable of unit time access to a shared queue of task descriptors and shared access to a common data structure. It is the task of the grid implementation to mimic the behaviour of this idealised machine as efficiently as possible.

The implementation proposed in this chapter adopts a straightforward approach. The absence of any form of real shared storage forces elements of the queue and data structure to be distributed across the grid processors' local memories. Each grid processor has a copy of the generic task code and performs the role of one idealised processor\(^1\). Thus, it simulates a sequence of idealised steps involving execution of accesses to the queue, to the shared data and of

---

\(^1\)Since the user is unaware of the number of processors in operation, there is nothing to be gained by considering a many to one mapping of idealised to real processors.
local instructions. Clearly execution of instructions of the first two types may involve references to objects stored non-locally thereby involving the cooperation of other processors. In order to avoid the unpredictable pattern of events which this might imply, the implementation places some structure on the way in which this cooperation takes place.

At each idealised time step a selection of accesses, at most one per processor, to queue and data will be required. In the proposed implementation these are separated into three categories, write access to the queue, read access from the queue and any access to the shared data structure. All grid processors (whether directly involved or not) are required to cooperate in the simulation of these accesses, before proceeding to the next idealised time step. Each category is simulated in a distinct phase, in the order as listed. Ensuing sections present algorithms which can implement the requirements of each phase.

The central operation in all three phases involves the routing of data objects around the grid. For the algorithms proposed, it is easy to calculate an exact bound on the number of processor to processor “pulses” of information required for their completion. Therefore, by counting the number of pulses (including empty “dummies” if necessary) processors can maintain synchronisation within and between phases without any explicit central control. In this way the simulation of each idealised step incurs a uniform slow down over the behaviour of the idealised machine. Section 4.3 discusses algorithms which could be used to implement the two phases involving access to the task descriptor queue, while section 4.4 deals with simulation of accesses to the shared data structure. A \( \sqrt{n} \times \sqrt{n} \) processor square grid is assumed throughout.

Firstly, several useful algorithms are noted, which will subsequently be used as building blocks.
BEGIN

place own packet on appropriate queue
FOR \(3(n^{*1/2}) - 3\) iterations DO BEGIN

PAR output one packet from each queue
{possibly including dummy packets for synchronisation}
PAR receive \(<=1\) packet on each link
WITH each packet received DO BEGIN

IF arrived at destination THEN write to task queue
ELSE add to appropriate link queue

END

END

END

Figure 4–2: A Routing Algorithm

4.2.1 Some Useful Algorithms

Sorting

Theorem 3 There exist grid algorithms which allow \(n\) fixed size items, initially distributed one per grid processor, to be sorted into row-major order across the grid in \(O(\sqrt{n})\) time. For example, such an algorithm is presented by Thompson & Kung [64].

Routing

Theorem 4 There exist grid algorithms which can route up to \(n\) fixed size packets across the grid, from unique sources to unique destinations (i.e. at most one
per processor) in no more than $3\sqrt{n} - 3$ routing steps with only constant time computation between steps.

Proof: Such an algorithm is presented. It uses the strict "row then column" routing strategy presented by Valiant & Brebner [71]. Packets are tagged with the unique identifier of the receiving processor. Since this identifier is unchanged throughout operation each processor may determine the relative grid location (row, column) of any other simply from its identifier. A packet is routed to its destination along a two branch path, first moving it along its original row to its final column and then up or down that column to the correct row. Any such path is of length at most $2\sqrt{n} - 2$. Each processor maintains queues for tasks requiring to be output on each of its four links and executes a copy of the code shown in figure 4-2. The justification for the $3\sqrt{n} - 3$ iterations results from lemma 4, which shows that at most $\sqrt{n} - 1$ more steps are sufficient on top of the $2\sqrt{n} - 2$ possible path length steps to ensure that each packet reaches its destination.

Note that no explicit synchronization is necessary to ensure that all processors know that the phase has terminated. It is sufficient to simply wait until $3\sqrt{n} - 3$ pulses on the links have elapsed.

Lemma 4 No packet suffers delays of more than $\sqrt{n} - 1$ steps on its route.

Proof: A packet will only be delayed if it has to join a non-empty queue to leave a processor on a particular link. No such queue can be encountered when travelling along the row, since routing in each direction along any particular row will be completely pipelined. Thus a packet may only be delayed when moving to its correct position in the destination column.

For any particular column, there are at most $\sqrt{n}$ packets which will arrive in it requiring to travel in one particular direction. In entering the "pipeline"
in a particular direction, each such arrival may hold up the flow through the column of all subsequent packets travelling in the same direction by at most one time step. Thus, the last arrival may be held up for at most $\sqrt{n} - 1$ time steps.

**Augmented Routing**

**Theorem 5** The routing algorithm described above will successfully route up to $n$ packets from unique sources to destinations shared by no more than $k$ (constant) packets in $(k + 2)\sqrt{n} - 3$ steps.

**Proof:** As above, noting that there may now be up to $k\sqrt{n}$ packets per column producing the possibility of a correspondingly longer delay moving up or down columns.

**Sum & Broadcast**

**Theorem 6** There exist algorithms which, given an $n$ processor grid with each processor storing some fixed size numerical value, sum the values and notify each processor of the resultant total in $4(\sqrt{n} - 1)$ steps.

**Proof:** One such algorithm is presented. Row totals are summed concurrently – the leftmost processor passes its value to its right hand neighbour which adds its own value and passes the result on. This is repeated along the row until the rightmost processor stores the row total. Obviously this is completed in $\sqrt{n} - 1$ steps. A similar operation is performed up the rightmost column, leaving the overall total stored in the upper rightmost processor. This total is then distributed back down the rightmost column and simultaneously back along the rows. Again each
of these phases involves $\sqrt{n} - 1$ steps, producing the required result.

4.3 Implementing the Queue

In this section we consider the implementation of a task queue controlled by a variety of disciplines.

4.3.1 The Stack Discipline

In the first discipline to be considered, the queue is operated as a stack (i.e. on a Last-In First-Out basis). At each idealised step the $p$ new tasks are placed at the head of the queue (in arbitrary order amongst themselves) and the $r$ requests are then served from the new head of the queue backwards. Thus if $r \leq p$ then the requests will all be served with tasks just placed on the queue. Otherwise, some processors will receive tasks inserted at previous steps. There is no guarantee as to which processors will be allocated the newer tasks, but it is essential that the set of allocated tasks is the newest possible.

Implementation requires some algorithm which can effect this idealised behaviour. This algorithm will be executed at every idealised step and its run time will represent the slow down factor incurred by the grid over the idealised skeleton. Thus, it is important to make it as efficient as possible. One such algorithm is now presented. This is introduced in the context of a shared memory parallel machine. Subsequently, its adaption to the grid is considered and a possible implementation is presented. Its performance is shown to be asymptotically optimal for the $n$ processor grid.
Implementing the Stack on a Shared Memory Machine

Before each idealized step to be simulated, elements of the task stack are stored in successive locations of an area of shared memory set aside for them. The location with the highest address in use is occupied by the head of the stack, and so on. A further shared memory location holds a pointer to the current head of the stack. To simulate the stack accesses of a particular step, the algorithm must deal correctly with the $p$ requests to place tasks on the stack and the $r$ requests to remove tasks from it, adjusting the stack and head pointer appropriately. It can be assumed that for the duration of the entire skeleton execution each processor is assigned a unique identifier between 0 and $n - 1$.

The algorithm has two phases, dealing respectively with addition and deletion of tasks to and from the queue.

In the first phase the $p$ "active" processors requiring to place tasks on the stack do so. To conform to the queuing discipline, these new tasks must be written into locations "head + 1" ... "head + $p$". The writing is performed directly by the active processors. In order to avoid clashes, the active processor with the smallest unique identifier will write its new task to location "head + 1", that with second smallest identifier to location "head + 2", and so on. Once the tasks have been written, processor $n - 1$ (whether active or not) will increment the head pointer by $p$, thus returning the stack to its stable state.

Given the use of a shared memory, the actual writing operations are trivial. The only problem lies in ensuring that each processor knows how many processors with lower identifiers intend to access the queue during the phase. This allows active processors to correctly identify the unique location to which they will write their new task, and allows processor $n - 1$ to update the head pointer appropriately.
Chapter 4. The Task Queue Skeleton

IF stack access required THEN Off[ID] := 1 ELSE Off[ID] := 0;
FOR j := 0 TO \log n/2 \ DO {base 2 \log of n/2}\n  IF jth bit of ID is set THEN
    Off[ID] := Off[ID] + Off[highest ID of lower group];

Figure 4–3: The offset algorithm for processor “ID”

An informal description of this “offset generation” algorithm aids clarity. The processors are repeatedly split into groups each half the size of the original, with processors in one group all having higher unique identifiers than those in the lower group. Processors calculate their offsets within the sub-group and processors in the higher numbered group then add to their offset that of the processor with the highest identifier in the lower numbered group. The resulting quantity is their offset among this larger group. The groups of processors repeatedly reform and add, until finally the top half of the processors add the offset of the processor with identifier $\frac{n}{2} - 1$ to theirs, completing the calculation.

This effect is achieved by having each processor execute the program fragment shown in figure 4–3, in which $ID$ is the processor's unique identifier and “Off” is an array in shared memory whose $i^{th}$ location will be set to contain the offset from the head of the queue to be used by processor $i$ (if active). Thus, Off[$n - 1$] will contain the offset to be added to the head pointer to point it at the new head of the queue.

Figure 4–4 illustrates the case with $n = 16$. For each iteration of the loop, indexed by $j$, processors involved in updating array Off are marked with an asterisk. These are selected by the conditional statement of the third line. A processor is involved during iteration $j$ if the binary representation of its unique
Chapter 4. The Task Queue Skeleton

Figure 4-4:

<table>
<thead>
<tr>
<th>count</th>
<th>( P_0 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
<th>( P_4 )</th>
<th>( P_5 )</th>
<th>( P_6 )</th>
<th>( P_7 )</th>
<th>( P_8 )</th>
<th>( P_9 )</th>
<th>( P_{10} )</th>
<th>( P_{11} )</th>
<th>( P_{12} )</th>
<th>( P_{13} )</th>
<th>( P_{14} )</th>
<th>( P_{15} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-5:

<table>
<thead>
<tr>
<th>count</th>
<th>( A_0 )</th>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( A_3 )</th>
<th>( A_4 )</th>
<th>( A_5 )</th>
<th>( A_6 )</th>
<th>( A_7 )</th>
<th>( A_8 )</th>
<th>( A_9 )</th>
<th>( A_{10} )</th>
<th>( A_{11} )</th>
<th>( A_{12} )</th>
<th>( A_{13} )</th>
<th>( A_{14} )</th>
<th>( A_{15} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

The identifier contains a 1 in the \( j \)th bit where the 0th bit is the least significant (try turning the figure through 90° clockwise).

Figure 4-5 presents an example of the evolving contents of the array Off during execution. The initial contents indicate which processors have declared their intent to access the queue during the phase, by writing a 1 to "their" element (processors 1, 5, 6, 7, 9, 11 and 14 in this example). Subsequent rows show the contents after each iteration of the \( j \) loop. For example, after the final iteration, processor 11 knows that it must write its task to stack location "head + 6". Similarly, processor 15 must adjust the head to allow for the 7 accesses made.

The second phase deals with the \( r \) requests to remove tasks from the stack. It may be implemented almost identically to the first phase. The offset calculation
Chapter 4. The Task Queue Skeleton

is implemented as before. An active processor $i$ will then remove a task from location “head – Off$[i]$ + 1” (the “+ 1” allowing the head itself to be removed). Finally processor $n - 1$ decrements the head pointer by the contents of Off$[n - 1]$, restoring the queue to its stable state.

Hillis and Steele [27] describe a similar algorithm, presented in the context of the SIMD virtual shared memory model of the Connection Machine.

Implementing the Stack on the Grid

A grid implementation of the algorithm presented above is now described. A simple technique for queue distribution and access is presented and a method of generating “offsets” from the head of the queue is introduced. Finally, the execution time of the complete algorithm is shown to be asymptotically optimal.

Storing the Queue

In the shared memory implementation the stack was stored in successive locations, allowing access to an element at any particular depth $d$ to be accessed by using $d$ as an offset to the head pointer. This technique is simulated in the grid implementation. An illusion of “successive locations” is created by having some pre-defined ordering on processors, known to all in advance. Successive locations from the stack area of shared memory are then assigned to the local memories of successive processors in the ordering, with wrap around from the last processor to the first. Thus the $i^{th}$ processor memory will hold locations $i, i + n, i + 2n$, etc.

\[\text{There is no need for adjacent processors in the ordering to be physically adjacent in the grid. The ordering may be completely arbitrary, the only important point being that it remains fixed throughout.}\]
Chapter 4. The Task Queue Skeleton

Additionally, each processor maintains a local copy of the "queue head pointer". Thus, a processor may calculate the location on the grid of any element of the stack, at any time, by reference to its own copies of the head pointer and the pre-defined ordering.

With this scheme in a stable state, the queued tasks are distributed evenly between the $n$ local memories. It also allows easy access to the queue for addition and removal of tasks. Having generated its unique offset (as discussed below) a trivial calculation (involving its copy of the head pointer and the ordering) allows an active processor to locate the processor whose memory is responsible for the relevant queue location. It is easily shown that this is the only access required of this target processor in the current phase.

**Theorem 7** Each processor will be responsible for at most one accessed queue location in each phase.

**Proof**: The locations have been distributed cyclically across all $n$ leaf processors, and so the first $n$ stack locations from the head up (in the case of adding new tasks) or down (when removing elements) are all located in different processors. Since no more than $n$ accesses can occur in any phase, and will always refer to locations within $n$ of the current stack head, these must always be located in distinct processor memories. •

Generation of Offsets on the Grid

The technique presented is based on the sum and broadcast algorithm of section 4.2.1. Processors are assigned unique identifiers in row-major order, as illustrated in figure 4-6. The offset generation problem is similar to that of section 4.2.1. Here, rather than accumulating the total of initial set values (all 0 or
Chapter 4. The Task Queue Skeleton

1) over the whole grid, each processor must know the sum of all values originating in processors with lower identifiers. Reference to figure 4–6 reveals that these processors are exactly those which appear to its left in its own row together with all those in rows above it. This can be exploited to obtain a simple $O(\sqrt{n})$ two phase algorithm.

In the first phase, a single packet is created in the leftmost processor of each row. This contains the initial offset (ie. 0 or 1) for that processor. Each packet is passed along its row. On receipt of such a packet, a processor adds the contents to its own initial offset and overwrites the packet with this new value. Thus, in $\sqrt{n} - 1$ steps every processor has accumulated its offset within its own row and the rightmost processor in each row knows the total offset introduced by that row.

In the second phase these row totals are distributed appropriately. A single packet is created in the top right hand processor and passed all the way down the right hand column. Each processor handling the packet (including the top right hand processor) adds its own row total to the packet, before passing it on down the column. It then creates a new packet, containing the original contents of the packet received down the column (which is just the sum of offsets from the rows above), and passes this to its left neighbour. This new packet is passed back along the row and added to each processor’s offset.
Chapter 4. The Task Queue Skeleton

The total time taken by the new offset algorithm is simply that necessary for a packet to move from the top left processor to the top right then to the bottom right and finally to the bottom left. This involves $3\sqrt{n} - 3$ links. Since there will be no collisions to cause delays and no more than constant computation at each node, the total time for the new offset algorithm is $\Theta(\sqrt{n})$.

Accessing the Queue

Suppose the current phase involves adding tasks to the queue. Then, with each processor having located its target memory, the problem reduces to that of routing a collection of packets across the grid. Since each processor can generate (or in the second phase, request) at most one packet, it is known in advance that all sources are unique and, by theorem 7, so are all destinations. Thus, the algorithm of section 4.2.1 can be used to achieve the routing in no more than $3\sqrt{n} - 3$ moves. Finally, the new head pointer is generated by processor $n - 1$ and is broadcast to the other processors using the broadcast phase of the sum and broadcast algorithm of section 4.2.1.

As before, the procedure for simulating a phase in which elements are removed from the queue is very similar. Here, two runs of the routing algorithm are employed. The first distributes requests for tasks and the second routes these tasks back to the processors requiring them.

It has already been shown that the offset generation can be performed in $\Theta(\sqrt{n})$ time on the grid. Thus, complete simulation of each idealised step can be performed in $\Theta(\sqrt{n})$ time. By lemma 5, this is asymptotically optimal.

Lemma 5 There is a lower bound of $\Omega(\sqrt{n})$ on the slow down introduced by any algorithm for the $n$ processor grid which correctly simulates the behaviour of an $n$ processor idealised machine using the stack discipline task queue.
Chapter 4. The Task Queue Skeleton

Proof: Consider two consecutive idealised time steps. In the first, exactly one processor writes a new task to the queue and no processors remove tasks. In the second no processors add tasks and exactly one processor removes a task. If the stack discipline is to be maintained correctly, then this must be the task added during the previous step. Such a sequence can occur between any two processors. In particular, it can occur in the grid between two processors which are $\Theta(\sqrt{n})$ apart and will thus require $\Omega(\sqrt{n})$ time simply to route the task across the grid.

4.3.2 A FIFO Queue

The algorithm implementing the stack discipline may be easily adapted to provide a first-in first-out queue. Here, new tasks are added to the end of the queue, while requests are served from the head. Thus, pointers to both ends must be maintained. The existing offset algorithm can still be used during task addition and removal, now referring to the appropriate end of the queue.

4.3.3 An Unordered Heap

In this queuing discipline no importance is attached to the order in which tasks are executed. New tasks may be placed anywhere and requests may be served with any available task descriptor.

As with the stack discipline, the problem with grid implementation is the likelihood that tasks will not be produced where they are needed. With a grid of $n$ processors a task produced during one step may be required by a processor $\Theta(\sqrt{n})$ links away in the next step. Thus, if it must be ensured that every request is served before computation may proceed, a $\Theta(\sqrt{n})$ slow-down must be expected for simulation of each idealized step. This is just the simulation penalty
incurred by the stack algorithm described above, which could therefore be used to implement this specification of the heap.

However, the heap discipline allows more freedom than the stack in the order of task evaluation. Its correctness would not be invalidated by allowing a task requesting processor to wait idle for a few idealised steps. Such processors could wait for suitable tasks to be produced in their "local" area of the grid, rather than holding up the entire computation for $\Theta(\sqrt{n})$ time to be served from a more distant processor. A trade-off appears to exist between time spent idling and time saved by abandoning grid-wide task routing. An algorithm which can be parameterised to cover this range is now considered.

Implementing Heaps

The algorithm proposed here attempts to exploit localised generation and consumption of tasks, at the expense of allowing a certain amount of processor idling. Essentially, the grid is divided into a large number of smaller, independent grids. For any idealised step, task generation and consumption is kept local to each sub-grid. Between idealised steps, surplus tasks may be shunted between neighbouring sub-grids. Thus, surplus tasks will eventually migrate around the whole grid. Meanwhile, the simulation slow down is reduced from $\Theta(\sqrt{n})$ to a function of the size of the sub-grids.

Behaviour of the algorithm is adjusted by a parameter $l$, which may take any value between 1 and $\sqrt{n}$ such that it divides $\sqrt{n}$ exactly, and is even or exactly $\sqrt{n}$. Thus, the grid may be divided exactly into $\frac{n}{l^2}$ square sub-grids of side length $l$. The balance between localised activity and global task distribution is maintained by having each sub-grid keep a local heap queue (using a local copy of the stack algorithm), while passing surplus tasks to other sub-grids. To this end, there is some pre-defined ordering on the sub-grids with the property that each sub-grid
Chapter 4. The Task Queue Skeleton

is between two sub-grids in the ordering to which it is physically adjacent (hence, by theorem 11, the requirement that \( l \) be even).

For any particular \( l \), the algorithm deals with the required queue accesses in three phases. Identical copies of the algorithm are executed by each sub-grid.

In the first phase tasks are added to and taken from the local queue, using the localised stack algorithm, in \( \Theta(l) \) time. By keeping track of the movements of the head pointer, each processor can know exactly how many tasks are on its local queue. Thus, if a processor wishes to remove a task but calculates an offset greater than the number of tasks which it knows to be locally available then it will postpone the attempt. It is possible that there may not be enough tasks on the queue in any particular sub-grid to satisfy all requests, even though there may be enough tasks globally.

In the second phase some surplus tasks (i.e. tasks remaining unused on the local queues), at most one per processor, are passed to the next sub-grid in the ordering. Since this sub-grid is a physical neighbour, these surplus tasks may be shunted simultaneously (along rows or columns as appropriate) from the processor storing them in one grid, to the corresponding processor in the neighbouring grid, in time \( l \). Each processor in the sub-grid losing tasks (knowing automatically how many tasks will have been shunted) can then independently update its copies of the head pointer and queue length.

Finally, in the third phase, tasks which arrived from a neighbour in the second phase are added to the local heap, again using the localised stack algorithm, in time \( \Theta(l) \). Thus, the whole algorithm executes in time \( \Theta(l) \).
Chapter 4. The Task Queue Skeleton

The Trade-Off

At one extreme, setting \( l = \sqrt{n} \), there is only one "sub-grid", the whole grid itself. The first and third phases become redundant and the algorithm is reduced to the standard stack implementation, with \( \Theta(\sqrt{n}) \) slow down for each idealised step. No processor will be left idle while tasks are available anywhere else in the grid. It would appear desirable to compare the performance for other settings of \( l \) to this simple case. Unfortunately, this is not possible in general. Consider an instance of some task queue algorithm. Execution of the instance with different values of \( l \) may result in different patterns of idling (even with the same number of processors available in total). This in turn can affect the actual sequence of tasks executed and may even result in the generation of a different set of tasks. Thus, a comparison between the run times of the two executions is not particularly meaningful. However, we can describe a more restricted class of problem instances from which useful observations can be drawn.

The instances of interest are those which generate enough tasks, with sufficient regularity, to keep constantly busy all \( n \) processors of a grid implementing an \( l = \sqrt{n} \) heap. Additionally, the instances must eventually generate the same overall set of tasks (i.e. the same amount of work), independent of the value of \( l \) and the pattern of idling. The \( l = \sqrt{n} \) implementation will execute such "robust" instances with a slow down of \( \Theta(\sqrt{n}) \) time over the idealised specification. As a trivial example of an algorithm which is "robust" in this sense, consider independently summing the elements of \( n \) vectors to produce \( n \) single quantities. Each task adds one element to the appropriate accumulator and generates a task to add the next. Clearly the overall set of tasks generated is independent of the data involved and the order of execution.

As noted above, implementations using other values of \( l \) incur an immediate slow down factor of \( \Theta(l) \). However, to make a valid comparison, it is also nec-
Chapter 4. The Task Queue Skeleton

necessary to consider the effect of processor idling on execution time. Since this is an idiosyncratic property of each instance, it is extremely difficult to identify any useful, general bounds on performance. On the other hand, it is possible to isolate particular (if artificial) instances which provide lower bounds on the worst case slow down.

As an example, consider the heap maintenance algorithm parameterised with \( l = 1 \). In this case, the second phase is redundant and the slow down introduced is \( \Theta(1) \) (i.e. constant). An instance of a problem is now sketched, for which the slow down due to idling can be as bad as \( \Theta(\sqrt{n}) \), thus nullifying the benefits of the fast heap handling.

Imagine a problem instance for which, at every idealised time step, \( \sqrt{n} \) idealised processors each produce new tasks which will each last for \( \sqrt{n} - 1 \) time steps. These are consumed by the other \( n - \sqrt{n} \) processors. The tasks are sufficiently long to keep all processors of an \( l = \sqrt{n} \) implementation busy all the time. Assume that the instance is robust in the sense described above.

Now consider an implementation of the same algorithm on the \( n \) processor grid, using \( l = 1 \). Suppose that the \( \sqrt{n} \) task producing processors are located together in the ordering used to circulate tasks, as shown in figure 4-7, with surplus tasks moving right. Then only \( \sqrt{n} - 1 \) tasks cross the boundary every \( \sqrt{n} - 1 \) idealised time steps, although \( n - \sqrt{n} \) are actually required on the right hand side. Thus \( \Theta(n) \) processors are being kept idle and all the work is being done by \( \Theta(\sqrt{n}) \) processors. Since the instance is robust, these will be required to perform the same operations as the \( n \) processors in the \( l = \sqrt{n} \) implementation. Thus, their overall execution time must involve a slow down of \( \Omega(\sqrt{n}) \) over the idealised specification. This is the same performance as that produced by the \( l = \sqrt{n} \) implementation.
Chapter 4. The Task Queue Skeleton

4.3.4 The Strictly Ordered Queue

In the fourth queuing discipline to be considered it is assumed that the task descriptions contain some key field by which they must be sorted. Thus, \( r \) requests for new tasks must be served with tasks having the \( r \) most significant (according to the definition of order) keys currently in the queue. In the abstract specification such an operation is performed in unit time. The problem of simulating its behaviour on the grid is now investigated.

It is immediately obvious that any implementation of the strict queue on the grid (or in fact any other realistic machine) cannot match the performance of the abstract specification. The crucial feature of the idealised machine which produces this result is best illustrated by considering the single processor version. By the abstract specification, this would be able to extract the most significant element (i.e. task descriptor) from an arbitrarily large set, in constant time. Clearly this is unrealistic, and it can be concluded that any implementation must have a slow down at each step (with respect to the idealised machine) which is dependent upon the length of the task queue at the time. Thus, a general result relating implementation slow down purely to \( n \), the size of the grid, is not obtainable.
Chapter 4. The Task Queue Skeleton

since the queue may become arbitrarily long. However, for certain problems, it may be possible to argue that the length of the queue is bounded in some way and it is interesting to ask how well the $n$ processor grid could handle such cases. For example, it might be possible to show that the length of the queue is always $O(n)$.

An algorithm is now sketched which implements the strictly ordered queue on the grid. The algorithm can handle arbitrarily long queues but its performance will be considered for cases in which the length of the queue is bounded to be $\Theta(n)$.

The algorithm uses the grid sorting algorithm of section 4.2.1 as a sub-routine. Rather than maintaining sorted order by insertion and deletion using some conventional technique based on the manipulation of pointers, the algorithm simply re-sorts the entire queue from scratch. As in previous implementations, the queue will be distributed across the grid so that any $n$ consecutive elements in it are located in the local memories of different processors. This allows easy task removal, given that the order on processors in the queue may be pre-defined (as is discussed in section 4.3.1. The algorithm has four phases:

1. Each processor is made aware of the total number of elements now on the queue by applying the sum and broadcast algorithm of section 4.2.1. These are not necessarily located evenly across the grid. Since each processor knows beforehand its own number of elements, this phase is $\Theta(\sqrt{n})$ irrespective of queue length.

2. A redistribution technique (e.g. that presented in section 5.5.2) is used to balance the queue elements evenly between processors. Since each processor can have at most one surplus element, a single run of the $\Theta(\sqrt{n} \log(\sqrt{n}))$ algorithm (by theorem 13) is sufficient. The number of elements, $cn$ say (for
the cases in which we are interested), is augmented with dummy elements (of "infinitely" low significance) to make it up to $kn$ where $k$ is the least perfect square with $k \geq c$. Each processor will know independently whether or not it should introduce any such elements. This phase takes $O (\sqrt{n} \log (\sqrt{n}))$ time (assuming the $\Theta (n)$ bound on queue length). Each processor now has exactly $kn$ queue elements.

3. Next, each processor imagines itself to be responsible for $k$ virtual processors, each of which contains one queue element. The physical grid simulates the grid sorting technique of section 4.2.1 on this virtual grid. This may be performed with only constant overheads using a simple simulation, since the choice of $k$ ensures that the larger grid fits neatly onto the smaller grid. The time for this phase is $O (\sqrt{kn})$ which is just $O (\sqrt{n})$ assuming a queue length bounded as before. Elements have now been sorted in "row-major" order on the virtual grid. However, the first $n$ elements in the queue will typically not be distributed one per physical processor as is required.

4. Processors forget about the virtual grid and re-arrange the (already sorted) elements into the required order around the loop. For the bounded queue each processor has only $k$ elements. These may be redistributed by means of $k$ calls of the augmented packet routing algorithm of section 4.2.1. In successive calls, each processor sends its most significant remaining packet (from the sorted queue) to its correct processor with a note of its position in the queue (thus giving unique sources and up to $k$-way shared destinations in the restricted queues under consideration). Once again, this takes time $O (\sqrt{n})$.

The whole algorithm executes in $\Theta (\sqrt{n} \log (\sqrt{n}))$ given the restrictions on queue size. It is interesting to note that this discipline represents a generalisation
of which the previous three are special cases. The heap may be interpreted as an ordered queue with all keys identical. LIFO and FIFO queues are ordered queues with keys simply noting the time of insertion and with significance increasing and decreasing with key value respectively.

4.3.5 A Note on Termination

The abstract specification defines execution of the skeleton to have concluded once all idealised processors are idle and the task queue is empty. Such a check can easily be implemented with the addition of a final $\Theta(\sqrt{n})$ time step phase after simulation of accesses to the queue, using the standard sum and broadcast algorithm presented in section 4.5. Since the simulation already includes $\Theta(\sqrt{n})$ slow down, such an addition would only increase overall run time by a constant factor.

4.4 Implementing the Data Structure

In this section, some solutions to the problems surrounding the shared data structure component of the skeleton are considered. Section 4.1 presented the data structure from the user's point of view. The structure may be accessed by any task instantiation, without restriction and every operation upon it is guaranteed to be executed eventually. From the implementation angle, access to the data structure occupies some area of a virtual memory space. At every time step, the idealised machine may generate up to $n$ accesses to arbitrary locations. In the realistic machine these locations must be distributed in some fashion across the local memories of the grid processors. The simplest solution is taken here, distributing idealised memory locations evenly across the local memory locations, one-to-one.
Chapter 4. The Task Queue Skeleton

At face value this appears to run into a problem familiar from the discussion on shared memory simulation of chapter 1. All processors may simultaneously wish to access locations in the same local memory. In a strictly synchronised machine this implies a $\Theta(n)$ time slow down. Fortunately, we can exploit a feature of the skeleton to avoid this problem.

In a conventional shared memory simulation (e.g. [71]) all idealized updates from one step must be performed before the next step may begin. This property is guaranteed to the user of the idealistic machine and will usually be necessary to preserve correctness of an algorithm. With the task queue skeleton, we have already noted the non-determinism introduced into the task execution sequence by the unknown number of processors. It is impossible to determine the precise order of task execution without knowing the number of processors (with this and knowledge of the clash arbitration scheme we could determine the exact order). Thus, the contents of the data structure found by a particular task is dependent upon the number of processors in use, and is consequently non-deterministic from the user’s point of view. The only ordering of tasks which can be guaranteed is that a task $t'$ produced by a task $t$ will not begin execution until $t$ has completed its accesses to the data structure (simply by keeping all additions to the queue until the end of task execution).

Therefore, to correctly model the idealised machine, it is sufficient to ensure that all accesses are eventually performed. Only ordering between each task and its descendants, and between instructions within the same task need be maintained. This allows the consideration of implementation algorithms in which certain processors are not immediately successful with attempted accesses. An unsuccessful processor can simply repeat the attempt during the next idealised time step, while successful processors proceed immediately with the next instruction. As noted above, this is in contrast to tightly synchronised abstract machines in which successful processors must wait until all are ready for the next virtual step.
Chapter 4. The Task Queue Skeleton

As a result, the more relaxed task queue scheme has two advantages. Firstly, for virtual steps in which all processors require to access the same memory, the $\Theta(n)$ step slow-down encountered by all processors in strictly synchronised schemes is replaced by a scale of delays, ranging from zero for the first successful processor to $\Theta(n)$ for that which is successful last.

More importantly, this has the side effect that the commencement of the next virtual instruction is staggered across the processors of the physical machine. Typical statistics on the locality of reference in programs suggest that the most likely place for such a clash to occur is immediately after or close to another. In a strictly synchronised scheme, the same problems would then be repeated. On the other hand, the staggered start produced in the task queue processors would destroy this symmetry before it became a problem. Processors successful quickly for one instruction will move on to quick success at the next, thus reducing congestion for those processors "following behind". Thus, in complete contrast to the strictly synchronised approach, clashes during one instruction will reduce the likelihood of subsequent clashes. This behaviour would be enhanced by the use of a suitable low order interleaving scheme [32] for the virtual to physical memory mapping, resulting in adjacent virtual addresses being located in independent physical memories.

Section 4.4.1 now considers the structure of a suitable implementation algorithm and properties required of it, while section 4.4.2 presents some variations on its internal details.

4.4.1 The Structure of a Suitable Algorithm

The situation is similar to that of implementing the task queue. At any step of the idealised abstract machine up to $n$ processors may wish to access data structure locations. Thus, a similar type of algorithm is suitable. However, it is
no longer guaranteed that these accesses will target unique processors, or even unique locations.

Fortunately, the situation also differs from the task queue in that it is no longer necessary for all packets to be routed successfully before proceeding. Processors will execute a routing algorithm for a pre-determined number of transfer steps (sending dummy packets to maintain synchronisation if necessary). Thus, unsuccessful processors can simply try again at the next idealised time step. However, in such a situation it will be necessary to ensure that each processor is aware of its packet's fate.

A suitable routing algorithm proceeds in two phases. In the first, packets are dispatched from the source processors (executing the tasks) towards the destination processors (responsible for the relevant data items) — there may be any amount of overlap between these two sets. Routing of packets proceeds for a specified time $t_1$ known to all processors (some simple function of $n$). Some packets will arrive successfully at their destinations within the time limit. In fact any useful first phase must guarantee that at least one attempted access is successful, otherwise the whole machine could stick trying to execute the same idealized step indefinitely. Thus algorithms will be required to exhibit property 1.

**Property 1** For any access step, at least one access packet arrives successfully.

In the second phase, destination processors which have received successful packets dispatch "acknowledge" packets to the successful sending processors. In the case of "read" access, the acknowledgement packet contains the requested datum. Again this phase has a known time limit $t_2$. Thus any sending processor which has not received an acknowledgement within time $t_1 + t_2$ knows itself to be unsuccessful and must attempt the data structure access again. Senders receiving an "acknowledge" packet know that their attempt was successful and continue
with their next idealised instruction. In the first phase it was not important that all attempted accesses were successful, but only that at least one was. Similarly, in the second phase it is essential that at least one successful access is acknowledged successfully. However this rather weak guarantee would result in much work being unnecessarily duplicated – successful accesses would be repeated. Therefore, we require that useful algorithms should exhibit a stronger property:

**Property 2** *Every* acknowledge packet arrives successfully.*

It will be shown that this can be guaranteed without significantly affecting the time needed for the second phase, $t_2$.

### 4.4.2 Implementing the Phases

Some possible variations in the implementation of the two routing phases are now considered. One course of action dictates that only the first arrival at each destination should be successful. This would guarantee that the second phase (sending acknowledgements) involved unique sources and unique destinations. Using the routing algorithm of section 4.2.1, it is therefore possible to execute such a second phase in $3\sqrt{n} - 3$ steps while maintaining property 2.

Taking this course, it is noted that the best performance now achievable is that the whole algorithm should successfully implement as many accesses as there are unique destinations in a particular step. As noted, the standard algorithm successfully completes the second phase and we now consider the first phase.

---

3In any case, it seems that the number of successes per destination should be kept low since entry to each processor is restricted to four channels.
Lemma 6 Any first phase algorithm which guarantees property 1 must have \( t_1 \geq 2\sqrt{n} - 2 \)

Proof: Consider a step which involves only one access, for which the packet must traverse the whole grid. Then the shortest possible route involves using \( 2\sqrt{n} - 2 \) links.

It seems obvious to ask how suitable the standard algorithm (with time \( 3\sqrt{n} - 3 \)) is for use in the first phase. Suppose that for a particular step there are \( d \) distinct destinations. Then,

Theorem 8 The standard routing algorithm cannot guarantee \( d \) successful arrivals given \( d \) distinct destinations in the first phase.

Proof: Consider an instance in which the bottom left hand processor has a packet destined for the top right hand corner processor and all other processors have packets destined for the second top processor in the right hand column. Then there are two distinct destinations but the packet from the bottom left processor will not arrive within the time limit, since it will be held up by the \( \Theta(n) \) length queue heading for the processor beneath its destination.

This example is easily adjusted to provide a counter example to any \( \Theta(\sqrt{n}) \) algorithm in which “hopeless” packets (i.e. those which will not be successful) are not removed en route to their destination. However,

Theorem 9 Use of the standard routing algorithm in the first phase does maintain property 1.
Proof: Because of the pipelining effect along rows, every packet arrives in its final column in time $\leq \sqrt{n} - 1$. Consider any such column and the packets wishing to travel up (down) it. These are distributed across a subset of the column processors. Within this subset there is one processor which is the uppermost (lowermost). The first packet to leave this processor (during routing step $\sqrt{n}$) cannot therefore be delayed and so must arrive at its destination within a further $\sqrt{n} - 1$ steps, giving it a total successful journey length of at most $2\sqrt{n} - 2$ time steps, the minimum time needed by any suitable first phase.

It is interesting to ask if the standard algorithm can be adjusted to guarantee $d$ successes (given $d$ distinct destinations) within a reasonable time. One solution is to allow processors to remove obviously hopeless packets (i.e. those which have destinations towards which the processor has already forwarded a packet). It is easily seen that it will not be good enough just to check against the destination which the processor originally used itself – the example from the proof of theorem 8 may be adjusted to provide a counter example. An alternative would have each processor keep note of all destinations to which it has already forwarded packets during the current phase. Now,

**Theorem 10** The new scheme, updated to keep note of all destinations seen by each processor, guarantees $d$ successes given $d$ distinct destinations in time $\leq 3\sqrt{n} - 4$.

Proof: Again due to pipelining it is clear that after $\sqrt{n} - 1$ steps all packets (remaining) will have arrived in their destination column. Consider any such column. As before, operations in the two directions are independent. For either direction there are at most $\sqrt{n}$ packets in each processor. Call the exact number $p_0$. Then $p_t$ will be the number present at time $t$ after this point. At each subsequent step the processor outputs at most one packet and reads in at most
Chapter 4. The Task Queue Skeleton

one (since there will be no more arrivals from along the row). The new packet will be kept only if it has a previously unseen destination.

In order that $p_t$ does not fall at a particular step (unless it is already zero) it is necessary that the new packet has a previously unseen destination. Initially $p_0$ of the $\sqrt{n} - 1$ destinations in the column are seen, and there are at most $\sqrt{n} - 1 - p_0$ unseen destinations left to which the processor may be asked to route packets. Therefore, of the next $\sqrt{n} - 2$ steps only $\sqrt{n} - 1 - p_0$ may be such that $p_t$ does not fall. Thus $p$ will fall in $p_0 - 1$ of them (unless it is already zero) and after the $\sqrt{n} - 2$ steps, $p_t \leq 1$, (i.e. there will be at most one packet in the processor).

This argument applies equally to all processors so that over the whole grid each processor will contain at most one packet. These may complete their journeys, in pipeline through the columns, in a further $\sqrt{n} - 1$ steps giving a total journey time of $3\sqrt{n} - 4$ steps.

The above proof assumed that checking whether a destination has already been seen is a constant time operation (i.e. independent of $n$). Any scheme which makes this realistic would obviously involve substantial space overheads. For example, a straightforward look-up table would require $n$ entries to note whether each possible destination had already been seen. This significant space overhead could be reduced at the expense of an increase in execution time. For example, noting that no more than $2\sqrt{n} - 2$ destinations will actually be seen by each processor in a particular phase, it would be possible to use only $\Theta(\sqrt{n})$ space per processor and employ some standard technique (e.g. a 2-3 tree) with time overhead $\Theta(\log n)$ to access these.

Alternative implementations must note that any scheme employing en route packet removal must use deterministic routing. Without this, all packets for
some destination could be removed by a processor which had seen the destination before.

4.5 Summary

Sections 4.2, 4.3 and 4.4 discuss grid implementation of the abstract task queue skeleton specified in section 4.1. At any time step of the "idealised machine", processors executing tasks may require access to the task descriptor queue or to the shared data structure. The algorithms presented show how \( n \) processor grids can co-operate to achieve simulation of such an idealised step. In the first two phases, accesses to the task queue are satisfied and in the third, processors attempt to deal with outstanding references to the data structure.

Four queuing disciplines were considered, of which three (FIFO, LIFO and "heap") were shown to be achievable with \( \Theta(\sqrt{n}) \) slow down at each idealised time step. The rigid discipline of the strictly ordered queue makes it less suitable for practical implementation although a possible algorithm was sketched. This would produce \( \Theta(\sqrt{n} \log(\sqrt{n})) \) slow down for queues with length bounded by \( \Theta(n) \).

A \( \Theta(\sqrt{n}) \) time algorithm implementing access to the data structure was proposed, together with some variations on the details of its implementation. A choice exists between the simple variant guaranteeing at least one success per time step, and that guaranteeing \( d \) successes given \( d \) distinct destinations at the cost of an increase in space. A realistic choice between the two could only be made on the evidence of experiments with real examples on a real machine.

In general, the simplicity and brevity of the algorithms described (with the possible exception of that sketched for the strictly ordered queue) suggest that the
behaviour obtainable in practice would not incur serious constant time penalties with respect to the asymptotic results described.

4.6 Examples

We now present a selection of applications which can be specified in terms of the task queue skeleton. The flexibility of the implementation, in particular with respect to shared data structure access, and the fact that such behaviour is very dependent upon details of problem instances make precise estimates of performance extremely difficult. However, for each example we can note the relative magnitudes of task instances and consider any limits which might restrict the number of tasks in operation concurrently.

4.6.1 One to All Shortest Paths

This may be implemented directly from the algorithm presented in section 4.1, with one addition. The idealistic model of [20] allowed variables in the shared memory to be “locked” by any processor (i.e. all other processors were denied access until the variable was “unlocked” by the same processor). This allowed processors inspecting a shortest path variable to lock it while deciding whether or not to update it. Any update could be completed before unlocking. The proposed implementation does not provide this facility. Thus, it is possible to conceive of a situation in which two processors, inspecting some vertex from tasks centred on different neighbours, both decide to update its shortest path with with values \( v_1 \) and \( v_2 \) say, with \( v_1 > v_2 \). If the processor submitting value \( v_2 \) is successful first, then the other processor will subsequently overwrite this value with the (by now) incorrect \( v_1 \).
Fortunately a simple amendment to the algorithm solves this problem. In addition to checking whether new shortest paths exist from the vertex passed by parameter to each of the neighbours, the task procedure must also check whether the neighbours themselves offer even shorter paths to the “central” vertex (because the central vertex has been incorrectly updated). If so, then the central vertex must be updated and a new task generated to check it again. However, this new task is certain not to become available until its “parent” has completed access to the data structure and the correct value the “central vertex” will be written when the appropriate neighbour is inspected in this “extra” task instance.

With this addition, the algorithm presented in [20] provides a good example of the task queue skeleton in use. It also illustrates the importance of taking fully into account the non-deterministic properties of the skeleton.

For this example, the length of a task instance is clearly proportional to the degree of the vertex under consideration, while the number of tasks executing concurrently is entirely dependent upon the internal details of the graph under consideration.

### 4.6.2 Graph Reduction

As a second example we outline the way in which the task queue skeleton could be used to implement a graph reduction algorithm, of the type discussed in [15], for the evaluation of functional programs. The abstract graph reduction mechanism of [15] is presented as a collection of processors (each with a local copy of all the function definitions) operating upon a pool of packets representing the graph (which is simply a tree). Packets contain pointers to their parents and children. Graph reduction proceeds by repeatedly replacing expandable packets by new groups of packets representing their expansion according to the function definitions, together with the repeated simplification of packets whose children
are sufficiently defined. An example of the former could involve the replacement of a packet representing the operation “accumulate.sum 1..15” by a new packet denoting the operation “+” and its two children “accumulate.sum 1..7” and “accumulate.sum 8..15”. An example of the latter might be the replacement of a multiplication node and its children “10” and “11” with the single node “110”.

The evaluation scheme of [15] assumes that each node has an identifier which is unique in the graph during the node’s lifetime, that identifiers are inherited as one node replaces another (with new identifiers generated for any children created), and that no restriction is placed on the the number of arguments to functions. Packets in the pool may be “active” (ready for manipulation) or “sleeping” (requiring evaluation of children before simplification).

The scheme is easily adapted for specification as a task queue algorithm. The shared data structure is a simple array of packet descriptors, indexed directly by packet identifier. Descriptions on the task queue will each contain the identifier of an “active” packet. The task procedure performs the appropriate manipulations for the packet under its consideration. During expansion it overwrites the description of the function packet with its new operation and leaves it “sleeping”. It creates new packets for the children as defined by the function expansion and places their identifiers on the task queue for further consideration. As in [15] the children note their parent’s identifier in order to pass back “wake up” signals after their own simplification. A node is simplified by evaluating its operation with respect to the new simplified arguments. Its packet on the data structure is overwritten with the new contents and the appropriate argument field of its parent is marked as “ready”. If all arguments are now ready then the parents identifier can be placed back on the task queue for its own simplification. Note that the clean semantics of graph reduction mean that no damage can be done by tasks concurrently returning “wake ups” from children. Even placing two copies of the same identifier on the task queue would only result in a small amount of
wasted effort (since the second such task activated would find its work already done).

By choosing to expand the graph as a binary tree we can use a simple standard mapping into the array, whereby the sons of the node stored in element \( i \) are to be found in elements \( 2i \) and \( 2i + 1 \). This allows immediate, independent generation of new packet identifiers by tasks. Function nodes with more than two arguments may be represented by "internal" expansion to a tree with the correct number of leaves.

A simple "heap" discipline for the queue would satisfactorily model the process pool. Since each task deals with a single node, the typical task execution time will be short. As noted in chapter 1, the number of tasks generated concurrently is in principle unbounded, being totally dependent upon the specific functional program undergoing execution.

### 4.6.3 LU Matrix Decomposition

A set of \( n \) linear equations in \( x_1, \ldots, x_n \) may be described by the equation

\[
Ax = b
\]

where \( A \) is an \( n \times n \) coefficient matrix, \( x \) is the vector \((x_1, \ldots, x_n)\) and \( b \) is a vector of constants. One method of solution requires that \( A \) be expressed as the product of two \( n \times n \) matrices \( L \) and \( U \), where \( L \) is unit lower triangular and \( U \) is upper triangular (see [76] for a thorough discussion). Overall solution is then simplified to the task of solving

\[
Ly = b
\]

to obtain an intermediate vector \( y \), then solving

\[
Ux = y
\]
Chapter 4. The Task Queue Skeleton

\[ l_{r1}u_{11} = a_{r1} \]
\[ l_{r1}u_{12} + l_{r2}u_{22} = a_{r2} \]
\[ \vdots \]
\[ l_{r1}u_{1r} + l_{r2}u_{2r} + \ldots + l_{rr}u_{rr} = a_{rr} \]
\[ l_{r1}u_{1,r+1} + l_{r2}u_{2,r+1} + \ldots + l_{rr}u_{r,r+1} = a_{r,r+1} \]
\[ \vdots \]
\[ l_{r1}u_{1n} + l_{r2}u_{2n} + \ldots + l_{rr}u_{rn} = a_{rn} \]

Figure 4–8: Producing the \( r^{th} \) row of \( L \) and \( U \)

to obtain the overall solution \( x \). Since \( L \) and \( U \) are triangular, both intermediate and final solutions can be obtained by simple back-substitution. This technique has the useful property that once found, \( L \) and \( U \) can be repeatedly used (without recalculation) to solve

\[ Ax = b' \]

for other constant vectors \( b' \). We now present a task queue implementation of an algorithm which generates \( L \) and \( U \) for a given \( A \).

The algorithm follows the method of [76]. The first \( r - 1 \) equations of figure 4–8 specify the interesting (i.e. not automatically 0 by triangularity) elements of the \( r^{th} \) rows of \( L \) uniquely, given the elements of the first \( r - 1 \) rows of \( L \) and \( U \). To generate \( l_{rc} \), \( r < c \), subtract all the appropriate product terms from \( a_{rc} \) and divide by \( u_{cc} \).

Similarly, the remaining \( n - r \) equations specify \( u_{rr}, \ldots, u_{rn} \) uniquely given the first \( r - 1 \) rows of \( U \) and the first \( r \) rows of \( L \). To generate \( u_{rc} \), \( r \geq c \), subtract the appropriate product terms from \( a_{rc} \). Division by \( l_{rr} \) is not necessary, since this is 1 by definition.
The simple induction which shows the method to be correct is based on the observation that \( l_{11} = 1 \) and that \( u_{ii} = a_{ii}, (1 \leq i \leq n) \) as a consequence. These define the first row of \( L \) and \( R \).

The task queue implementation generates \( r \) tasks during the calculation of the \( r^{th} \) row of \( L \) and \( U \). The \( i^{th} \) task calculates \( l_{ri} \) by division then performs all the products and subtractions involving this final value. The first task of the \((r + 1)^{st}\) row is generated once the final task of the \( r^{th} \) has completed.

A possible coding of the generic task is illustrated in figure 4–9. Noting that the location initially storing \( a_{rc} \) can be directly transformed (by subtraction and division) into \( l_{rc} \) (if \( r > c \)) or \( u_{rc} \) (otherwise), it is clear that no extra memory space need be set aside for \( L \) and \( U \). Strict progression through the operations associated with each row ensures that \( a_{rc} \) will have been transformed into \( l_{rc} \) or \( u_{rc} \) before it is accessed as such. Thus the shared data structure required is a two dimensional array, initially storing the elements of \( A \) accessed by row and column number. Additionally, an \( n \times n \) element integer array “mark” will be required and is introduced below.

The task queue stores pairs of integers. The pair \((r,c)\) specifies the task which calculates \( l_{rc} \) by dividing \( a_{rc} \) by \( a_{cc} \) (which is now equal to \( u_{cc} \)). Elements of two dimensional array “mark” are initialised to zero and indicate the number of subtractions which have been performed upon the corresponding element of \( A \). This is used to schedule division and the generation of the first task of the next row. The tasks here are more substantial than those in the two preceding examples. However, as we have noted, generation of the \( r^{th} \) row produces a maximum concurrency of \( r \) tasks, and all tasks from one row must terminate before any from the next may begin. This suggests that this implementation will be most effective when \( n \) (and hence many values of \( r \)) is much larger than the number of processors available.
TASK generateL (r,c : integer)
VAR temp : real; {a local variable}
BEGIN
WHILE mark[r,c] <> c-1 DO ; {wait for relevant subtractions}
A[r,c] := A[r,c]/A[c,c]; {i.e. L[r,c] := A[r,c]/U[c,c]}
temp := A[r,c] * A[c,c+i]; {i.e. L[r,c]*U[c,c+1], to be}
{subtracted from A[r,c]}
A[r,c+1] SUB temp; {indivisible subtraction}
mark[r,c+1]ADD 1; {indivisible addition}
IF c+1 < r THEN put_task(r,c+1); {task generating L[r,c+1]}
FOR i := c+2 TO n DO BEGIN {subtract appropriate products}
temp := A[r,c]*A[c,i]; {i.e. L[r,c]*U[c,i] to be subtracted}
{from A[r,i]}
A[r,i] SUB temp;
IF (i<r) OR (i=n) THEN mark[r,i] ADD 1;
END;
END;
END;

Figure 4-9: Code for the LU decomposition task
4.6.4 Solving a Band Matrix System of Linear Equations

McKeown [44] discusses a parallel implementation of an asynchronous iterative scheme for solving systems of linear equations where the co-efficient matrix is a band matrix. The method is easily formulated as a task queue algorithm.

A succession of approximations to the $n$ element solution vector $x$ are generated with each element of each approximation computed as a function of previous approximations and the constant elements of $A$ and $b$. The crucial feature is that eventual success (i.e. convergence of the approximations to the actual solution) does not require the generation of successive approximations to each element of $x$ to proceed synchronously. It is sufficient to use the most recent available approximation to the other elements, provided that these also progress eventually. Thus, the lack of a guaranteed ordering of updates of the shared data present in the task queue skeleton is acceptable.

The shared data structure is simply a vector of real numbers, one for each element of $x$. Depending upon space available, the constants $A$ and $b$ could be either shared or copied locally to each processor. Items on the task queue will be integers in the range $1..n$ and a task given parameter $i$ generates a new value for $x(i)$ using the current elements of $x$ in its calculation. On completion it writes the new value of $x(i)$ and generates a new task descriptor $i$. Use of a FIFO queuing discipline will ensure that generation of new values proceeds reasonably evenly across $x$, thereby aiding convergence. The queue initially contains one descriptor for each element of $x$. A limit to the number of iterations could be set by an additional task parameter, noting the number of times that a particular element has been recomputed. Generation of a new task descriptor for that $i$ would depend upon the new parameter not having exceeded the limit.

Again, the average task length is substantial (involving inspection of all $n$
elements in the current solution). Maximum concurrency is bounded by \( n \), but synchronisation of the kind encountered in the preceding example is not required.
Chapter 5

The Iterative Combination Skeleton

The skeleton presented in this chapter deals with a more specialised collection of algorithms than the task queue skeleton of chapter 4. In common with the recursive divide and conquer skeleton of chapter 3 its abstract specification is purely sequential.

5.1 Abstract Specification

5.1.1 Introduction

Many problems may be solved by algorithms which progressively impose structure onto an initially uncoordinated collection of objects. Typically, an instance of such a problem is described by a set of homogeneous objects, together with details of any relevant internal structure of each and of any relationships existing between them. Given a rule for combining or in some way relating two objects, and a measure of the value of the combination, the algorithm iterates through a loop in which each object is combined with the most suitable remaining other object, if such exists. The loop is repeated until either all objects have been
combined into one (i.e. the complete required structure has been imposed), or no further acceptable combinations exist. Such algorithms are often classed together as "greedy" [1], by virtue of the locally rather than globally optimal nature of decisions made. This policy may or may not lead to globally optimal results, depending upon other features of the specific problem in hand.

As a concrete example, consider Sollin's algorithm [8] to compute the minimum spanning tree of an undirected graph. The objects describing an instance are all subtrees of the graph under consideration. Initially there is one sub-tree for each vertex. Two objects are combined by finding the shortest edge which joins them - the cost of such a combination is simply the cost of this edge (or infinitely large if no such edge exists). The best partner for an object is that with which it may be combined at lowest cost, i.e. the remaining sub-tree to which it may be joined by an edge of lowest cost.

At each iteration, for each remaining sub-tree, the algorithm finds the edge of lowest cost which joins the sub-tree to some other sub-tree. These trees are replaced by a new tree representing the old trees joined by the short edge. The process is repeated until only one tree remains (the minimum cost spanning tree of the original graph), or until the remaining sub-trees have no edges between them. In this case, no combinations can be found and the trees represent a spanning forest of the original, unconnected graph. The "iterative combination" skeleton models this style of solution. Its abstract specification and possible parallel implementation on the grid are presented in this chapter.

5.1.2 Specification

To the user, the "iterative combination" skeleton is seen to implement the "pseudocode" presented in figure 5–1, where S represents a set of homogeneous objects describing a problem instance.
WHILE |S|<>1 AND NOT failure to find any combinations DO BEGIN
    FOR each s in S
        find s' in S s.t. s' is the "best" partner for s
    FOR each s in S DO BEGIN
        combine s and s' to form s''
        remove s and s' from S and add s''
    END
END

Figure 5–1: The Abstract Specification

In order to tailor the skeleton to a particular problem, the user must provide a type specification of an "object", a collection of such objects comprising the initial set S, details of a procedure which allows two objects to be combined and returns a measure of the cost of such a combination and finally, a method of discriminating between two such costs to select the more desirable.

It is not difficult to conceive of examples in which the best partner relationship is not symmetric, i.e. in which an object s₁ may be the best partner for s₂ but have s₃ as its own best partner. In such examples, all three objects involved in the "group" must be combined and replaced by a single object. For this reason, the combination procedure will be assumed to be both commutative and associative, i.e. the result of combining a particular group of objects at some iteration is independent of the order in which the objects are actually combined. This does not appear to be an unnatural restriction. The problem does not arise in examples where best partnership is symmetric.
5.2 A Parallel Implementation Technique

The parallel implementation discussed in this chapter attempts to exploit the most obvious sources of parallelism in the abstract specification, namely the two “for each s in S” loops. A sequential implementation would have to deal with objects in S one at a time, for both loops. Given as many processors as objects, a parallel implementation could, at each iteration, find all best partners concurrently then proceed to perform all appropriate combinations concurrently. The grid implementation proposed in this chapter takes such an approach.

The remainder of this section introduces the issues to be addressed in designing such an implementation. In subsequent sections, an actual grid implementation is proposed which addresses the problems raised.

5.2.1 Distribution of Valid Information

The membership of the set of objects changes from one iteration to the next - some objects disappear while other new ones are created. Any processor involved in finding and combining best partners during some iteration must only deal with up to date descriptions of the objects. Trying to combine with an object which should no longer exist would contradict the specification of the skeleton. Thus, after each iteration there must be some consistent and accessible representation of the objects currently in the set.

5.2.2 Testing and Selecting Partners

For a particular iteration it will be necessary to test all possible pairings of objects which involve at least one new member (i.e. created during the previous
iteration). In the most extreme cases this will involve comparing all pairs of remaining objects. Thus, each object may be the subject of many concurrent tests. Any parallel implementation must attempt to minimise the problems caused by this sharing of information, possibly by careful scheduling of access or by replication of data. Similarly the selection of a best partner will require a consensus to be reached between any processors which are sharing the workload.

5.2.3 Combining, Updating and Checking for Termination

Once the best partners have been selected for each object the appropriate combinations must be executed and the set of remaining objects adjusted accordingly. The possibility that multi-object combinations may occur complicates the situation. The implementation must ensure that only one new object is created, replacing all its components. After the combinations, a check must be made as to whether the algorithm should terminate. The most obvious solution involves noting how many objects remain and comparing this number with the number which remained after the previous iteration. The result must be known to all processors.

5.2.4 Balancing the Load

As a consequence of each iteration the number of remaining objects and their respective sizes are altered. Thus, the quantity (with respect to particular objects) and location of work (for some particular implementation) to be done during the next iteration will be affected. A parallel implementation should take into account the way in which this changing workload can affect performance. It may have to
make adjustments to the balance of work across the machine in order to preserve efficiency.

5.3 Implementation on an Idealised Grid

This section presents techniques which could be used to implement each iteration of the "test, select and combine", loop. These make the unrealistic assumptions that the number of objects present at the start of the current iteration exactly matches the number of grid processors, and that the side length of the grid (or, as will become clear, the appropriate sub-grid) is even. Section 5.4 considers the modifications necessary to move to a completely practical implementation on a fixed size grid where these assumptions do not necessarily hold.

The implementation is founded on the idea of assigning each object to a unique "home" processor for the duration of the current iteration. This processor is responsible for finding the object's most suitable partner and is the only one which keeps a record of the object's description throughout the iteration. In order to select the best partner it must see details of all the other remaining objects and test the desirability of combining with each of them. Thus, each processor is supplied with a local copy of the user specified procedures implementing object combination, costing and cost discrimination. Having selected the "best partner" it must co-operate with that object's home processor (and with any others involved in a group of combinations) in combining the objects and passing exactly one copy of the resulting object into the next iteration. The technique described below allows these functions to be carried out in a well-structured fashion, for which the required pattern of communication is independent of any particular problem instance.
Each object is assigned a unique integer identifier before execution of the skeleton. An object formed by combining a collection of others inherits the lowest identifier among those of its components. Note that these identifiers are generated and inspected by the implementation system only, being entirely independent of any identification introduced by the user.

5.3.1 The "Test and Select" Phases

Suppose that processors are connected only by a point-to-point ring. A very simple technique allows such a ring to implement the "test and select" phases of the skeleton.

Each processor makes a copy of the description of its home object, including all the information required to test combinations with it. These copies are passed round the ring in a pre-defined direction. Upon receipt of such a copy, a processor simulates the result of combining the new arrival with the static copy of its home object. Each processor keeps a note of the best combination it has tested so far in the current iteration. Once the copies have been passed right round the ring, each processor knows the identity of the best partner (if such exists) for its own object.

Assuming for the moment that all objects are of the same size, it can be seen that this method gives speed up to within a factor of two of optimal for the test and select phase. There are $\Theta(|S|)$ pairs and these are being tested $|S|$ at a time. Thus, provided that the time taken to test an object is at least as large (asymptotically) as the time taken to transmit its details from point to point$^1$,

---

$^1$This assumption seems reasonable - otherwise it would appear that some of the data transmitted is redundant.
communication will take up at most a constant fraction of the total time round the ring and so only restrict optimal speed up by this constant factor.

However, the goal of the exercise is to implement the skeleton on a grid of processors and the task of moving from ring to grid is now considered. Theorem 11 shows that the assumption that the grid side length is even is both necessary and sufficient to ensure that the move can be made efficiently. In such cases, ring processors may be mapped one-to-one to grid processors in such a way that neighbours in the ring are neighbours on the grid (i.e. that the ring marks a Hamiltonian circuit of the grid). Thus, there will be no time penalty associated with the simulation.

**Theorem 11** A Hamiltonian circuit of a square grid exists $\iff$ the side length of the grid is even.

**Proof:** (i) side length even $\implies$ tour exists

In such cases, a tour of the grid is easily described.

Start in the top left hand corner. Go up and down adjacent columns, avoiding the top row. Since there are an even number of columns, the tour has progressed to the top right hand corner and can be completed along the top row. Figure 5-2 illustrates the instance with side length six.
(ii) side length odd $\implies$ no tour exists

Imagine rows and columns to be numbered from 1 to $\sqrt{n}$ along and down, starting at the top left hand corner. When following a path through the grid, it is possible to keep a two bit state, where the bits note whether the current processor is in even or odd numbered rows and columns respectively.

Any move between directly connected processors flips exactly one bit of the state. Since a circuit must start and finish in the same processor and hence the same state, this implies an even number of moves. However, if the side length is odd then so is the total number of grid processors. A Hamiltonian circuit involves moving into each processor exactly once, implying an odd number of moves. The resulting contradiction proves that no such circuit exists in this case.

5.3.2 Combining Objects

To complete the work required in a particular iteration it remains to combine objects as required by the choice of best partners. Only one copy of each new object should be kept for the next iteration, and all copies of its components in their original form must be removed.

Before combination begins each processor has explicit knowledge of its object's direct best partner. It has no knowledge of any groups of combination with which it may be involved. Any implementation of the phase must allow for such situations. Two techniques are now presented which deal with the problem in a completely general way, making no attempt to tailor activity to suit specific patterns of combination. It is argued in section 5.6 that the simplicity of these approaches will lead to superiority over other methods, in all but the most trivial (and unlikely) of examples.
The Single Tour Algorithm

This method involves one further tour of the grid embedded processor circuit. Again, each processor places a copy of its home object onto the circuit. This copy is placed in a packet tagged with two extra fields. As the packet progresses, the first field notes the identifiers of objects which have been combined with the original object, during the tour. The second field notes the identifiers of objects with which the packet is aware that it must still combine. These are simply the "best partners" (except for those already incorporated into the packet) of the objects noted in the first field. A copy of the original packet is kept by the home processor.

On receipt of an incoming packet, a processor cross checks the fields to see if a combination should take place (on the strength of the available information) between the packets. If so it creates a new packet subsuming the two old packets with tag fields set accordingly. It replaces its old stored packet with the new one, before passing a copy of the new packet on round the ring. If no combination is discovered, the old incoming packet is passed on.

Theorem 12 proves that this scheme produces the required result - that on completion of the tour each processor has a copy of the packet representing the object into which its old object has been merged (just the old packet itself if no combinations were made). However, there will now be $c$ copies of each new object which subsumed $c$ old ones. All but one of these must be removed. The surviving copy is arbitrarily chosen to be that resident in the home processor of the component with the smallest identifier (which the new packet inherits in preparation for the next iteration). Since all processors know the identity of their original object, and those of all the components of the new resident packet (by inspecting the first field), redundant copies can be deleted concurrently, without further inter-processor consultation.
Theorem 12: The single tour algorithm ensures that all packets are transformed to represent the merged set of all objects with which their original object has been combined.

Proof: Consider any object and label it $o_1$ and its original home location $h_1$. Imagine the loop to be cut "behind" $h_1$ and stretched out to the right with all packets moving right and "wrapping round" from the end to $h_1$.

Consider any object $o_x$ with which $o_1$ (and hence its travelling packet) must be merged. Then there must always be a sequence of objects $o_1, o_2, ..., o_x$ such that $o_{i+1}$ is the "best partner" of $o_i$ for $1 \leq i \leq x$ (otherwise we wouldn't be combining $o_1$ and $o_x$). Objects in the sequence have original locations $h_1, h_2, ..., h_x$. Suppose that $h_k$ is the rightmost of these in the cut loop. It is now proved that the static packet in $h_k$ accumulates $o_2, ..., o_x$ before the copied packet containing $o_1$ arrives there and hence merges them with $o_1$ on its arrival.

This is achieved in two parts, by proving that

1. $h_k$ accumulates $o_2, ..., o_k$, and
2. $h_k$ accumulates $o_{k+1}, ..., o_x$.

(and that all this happens before $o_1$ has passed). Both parts may be proved inductively. A proof of part 1 is presented here and may be simply adapted to prove part 2.

As a base case for induction it is noted that $h_k$ accumulates $o_{k-1}$. The packet originating in $h_{k-1}$ must pass $h_k$, and will be recognised by the fact that its second field contains the identifier of $o_k$. Since $h_{k-1}$ is to the right of $h_1$ this will happen before the packet originating in $h_1$ arrives at $h_k$.

Given the inductive hypothesis that $o_j$ is accumulated at $h_k$, it is now proved that $o_{j-1}$ is also accumulated. This will hold for $3 \leq j \leq k - 1$. In conjunction
with the base case, proof of part 1 follows by induction on \( j \) from \( k - 1 \) down to 3.

Suppose that \( h_{j-1} \) is to the right of \( h_j \) in the cut loop. Then the packet which successfully took \( o_j \) to \( h_k \) must recognise and pick up \( o_{j-1} \) as it passes \( h_{j-1} \) before arriving at \( h_k \).

On the other hand, \( h_{j-1} \) may be to the left of \( h_j \). In this case, the packet originating in \( h_j \) will be recognised and accumulated at \( h_{j-1} \), and the resulting packet will also be recognised upon reaching \( h_k \), by virtue of containing \( o_j \).

The theorem is proved by the observation that the original choice of \( o_1 \) was arbitrary. Consequently, the argument applies to any packet.

The algorithm has the useful property that it requires only one further tour of the grid to implement the combination of objects, in any instance. However, the way in which multiple near-copies of increasingly large objects are moved around while under construction appears costly. An alternative combination algorithm is now presented. This algorithm involves three tours of the grid by constant size packets followed by one tour involving only objects present at the start of the iteration. New objects are constructed uniquely by the processor responsible for them at the start of the next iteration.

The Alternative Algorithm

The alternative algorithm proceeds in two phases. In the first phase, each processor is made aware of the lowest numbered object in the group with which its own object must be combined. In the second phase processors dispatch their object to the home of the identified object. The processor there combines all arrivals
Chapter 5. The Iterative Combination Skeleton

discarded edges

Figure 5–3: Savage’s Systolic Array

together with its own object to create the single copy of the new object required in the next iteration.

The proposed first phase is motivated by the observation that the problem of identifying the objects of lowest identifier in “combination groups” is a simple instance of the problem of finding the vertex of lowest identifier among each connected component of an undirected graph. Quite simply, objects correspond to vertices and “best partner” pairs to undirected edges. Thus, a direct implementation of a connected components algorithm will also solve the “lowest object identifier” problem.

Savage [57] presents a suitable, systolic connected components algorithm. This involves a linear array of processing elements, where processor $i$ is responsible for vertex $i$ of the input graph. Each processor calculates the identifier of the smallest numbered vertex in the component to which its own vertex belongs. This is achieved as depicted in figure 5–3, by pipelining constant sized packets containing descriptions of edges along the array and back, before being discarded.

As packets progress, their contents, and the information stored by processors, are updated, but their size remains constant. The edges are initially stored in an off-chip “pool”. Each edge passes along the array and back exactly once, with constant time between each pulse which moves them between neighbouring
new edges

Figure 5–4: Embedding the pool in the tour

processors. Thus, the whole algorithm takes time $\Theta (v + e)$, where $v$ is the number of vertices in the graph and $e$ the number of edges. Clearly the existing tour of the grid allows direct simulation of the linear array, with no more than constant overheads. To complete implementation, it is necessary to simulate the behaviour of the edge pool. This is required to feed edges into the simulated array at every second systolic “pulse”. To achieve this, it is noted that each processor in the tour is initially responsible for one “edge”. Thus the “edge pool” is already distributed around the tour. Furthermore, by virtue of the fact that the array is embedded in a circuit, the $n$th processor of the array is physically adjacent to the first. Thus the edges can be pipelined round the tour “entering” the simulated array over

---

2 Although Savage’s algorithm depicts the processors as being numbered in increasing order away from the “edge pool” it may easily be verified that this is not a necessary condition for the correctness of the algorithm. Thus, no re-numbering of objects need be implemented to produce this situation.
the link which joins the first processor to the \( n^{th} \), as suggested in figure 5–4. Since each processor is now responsible for both part of the array and part of the pool, simulation of movement in these two structures is interleaved. In total, each "edge packet" makes up to three tours of the grid: at most one to "get out" of the edge pool, one along the "processor array" and one back.

A further tour of the grid is now invoked to allow each processor to send its object to the home of that into which it has to be combined before the next iteration. Finally, new objects are produced by combination at these destination processors.

The cost of this last operation is crucial in determining whether this approach is superior to the first algorithm presented. This was vulnerable to iterations in which multiple copies of large objects were transported around the tour while under construction. This has been alleviated at the expense of introducing three additional tours by constant size packets. On the other hand, the large objects must still be constructed by the processors responsible for them at the next iteration. A reasonable decision as to the relative effectiveness of the two algorithms could only be made on the evidence of real examples on a real machine.

### 5.3.3 Checking for Termination

At this point the current iteration is complete. It now remains to check whether a further iteration is required, which involves counting the remaining objects. If the total number has either fallen to one, or has remained unchanged from the previous iteration, then execution is complete. In other cases another iteration must be initiated. The sum and broadcast algorithm of section 4.2.1 is suitable for generating and distributing the count and obviously takes insignificant time in comparison with the tours involved in the rest of the iteration.
5.4 Fixed Size Grids and Redistribution

The preceding sections presented an implementation for iterations in which the number of objects present is exactly equal to the number of processors in an even sided grid. Unfortunately, this match will occur only rarely in practice, and a realistic system must be able to handle the problems associated with its absence. There are two ways in which such a mismatch can occur, which will be discussed separately.

In the first (and given the fixed size of any real machine almost inevitable), the number of objects present initially is much larger than the number of processors. The system has to distribute the workload between processors both initially and dynamically, as the number of objects falls. This possibility is discussed first.

A second possibility, considered in section 5.5, is that the number of processors is as large as the initial number of objects. As the number of objects falls, the system may be able to improve performance by regrouping remaining objects into increasingly smaller areas of the grid. Of course, this situation will also occur in the final iterations of instances which initially fell into the first category.

5.4.1 Large Problems on Fixed Size Grids

In these problem instances the initial size of the set of objects, \( S_0 \) say, is assumed to be much larger than the size of the grid. For a square grid of \( n \) processors, \( S_0 \) may be arbitrarily large with respect to \( n \). Since \( n \) is now fixed (for any particular machine) it is not unreasonable to assume that \( \sqrt{n} \) is even (simply by ensuring that grids are physically constructed thus). A very simple implementation which deals with this situation is now presented, and its performance for certain examples is analysed. Although it is impossible to predict the "typical" use of the
Chapter 5. The Iterative Combination Skeleton

skeleton, these examples give helpful insight into the seriousness of difficulties which may arise.

5.4.2 A Simple Implementation

For an iteration involving $|S|$ objects, the implementation involves a simple simulation of the required $|S|$ long circuit of processors by the $n$ processors actually available. Before the first iteration each processor takes responsibility for $S_n$ (as nearly as possible) objects. These are assigned to adjacent homes in the length $S_0$ virtual circuit, and blocks of such homes are assigned to adjacent real processors in an order following the real circuit around the grid. The iteration is implemented by having each processor simulate one instruction from each of its resident virtual processors in turn. In this way, communication will only ever be required between real processors which are physically adjacent and therefore introduces no more than constant overheads for each message passed.

After each iteration, each processor may lose some of its objects. If this happens then it simply proceeds as before during the next iteration, now sharing its time between a smaller number of virtual processors. Should a processor lose all its objects before the final iteration, it will simply act as a write-through buffer, forwarding messages between its neighbours.

5.4.3 Examples and Analysis

The scheme proposed amounts to a fixed size pipeline simulating a variably longer pipeline. It is well known [32] that, in general, the significant factors which act to reduce pipeline efficiency are gaps in the flow of data and uneven length (in time) of stages. How and when do these feature in the simple implementation?
Since the real processors are always ready to adapt to a new smaller number of resident objects, the only way in which gaps can occur is for some processor to lose all its objects while others are still active. This would introduce a single time step delay into the pipeline. While isolated instances will not be significant, efficiency could be adversely affected if large numbers of processors quickly become idle.

The dangers of unbalanced stages appear more pressing and may appear in two ways. Firstly, as objects disappear from one iteration to the next, it seems probable that some processors will be left with more resident objects (and hence virtual processors) than others. These physical processors will tend to become bottlenecks in the physical pipeline. Secondly, as objects are combined (in an unpredictable and problem dependent way) it seems likely that certain of the remaining objects will grow larger than others. This will probably involve them in longer testing and combination time, and will certainly involve longer transfer time, leading to imbalance and bottlenecks between stages of the virtual pipeline.

Some examples are now presented which illustrate the way in which these issues can affect performance. While it is impossible to predict what constitutes a "typical" example, the situations discussed nevertheless provide a useful insight into the way in which such problems may interact.

Several assumptions have been made throughout the examples to ease analysis. Firstly, it is assumed that all objects in the initial set have descriptions of uniform size. The time taken to transfer or examine such a chunk of data is taken as the basic unit of time in the analyses. The combination of two objects of size $x$ and $y$ is assumed to produce an object of size $x + y$. Similarly, it is assumed that the time taken to combine two objects and test the cost (by whatever measure is interesting) of the new object is proportional to its size.

Each example analyses the cost of some particular pattern of combination. The best and worst possible scenarios (in terms of the balance of object distri-
bution at each iteration) are compared. The best scenario models the situation in which all remaining objects at each iteration are evenly balanced between processors. The worst scenario represents the case in which the balance is as uneven as possible, given the initial distribution. Comparison of the two gives some insight into the improvements which might be achieved by the introduction of an object redistribution algorithm between phases. Of course, to be useful, such an algorithm would have to take less time than the saving produced. We will be particularly interested in behaviour when the initial number of objects present, \( S_0 \), greatly outnumbers the number of processors available (for example, when \( S_0 = \Omega(n^2) \). There are two reasons for this. Firstly, a large number of objects ensures that sufficient work is involved to allow any serious inefficiencies to become apparent; Secondly, these cases reflect the realistic situation in which the size of an actual machine is constant, but the sizes of the problems are not.

A variety of techniques will be used to estimate the time taken by a particular iteration. All the estimates actually consider the time spent during the combination testing phases. Although the time spent on actual combination is excluded this will simply be some constant multiple of that spent testing (since both involve a single tour of the grid with similar objects). Thus, the estimates will be accurate to within a constant factor of the actual times. This is good enough to provide evidence for the conclusions drawn. If all processors are responsible for at least one object (i.e. there are no holes in the pipeline) then the measure is

\[
\text{Time} = \text{No. of objects in processor responsible for the largest object} \times \text{total No. of objects} \times \text{Size of largest object}
\]

This is justified by considering the time taken by the real processor responsible for simulating the virtual processor which is the home of the largest object during the phase. To complete the phase, this virtual processor must compare its (large)
object with all the other objects in the system. This introduces the second and third terms. (Strictly, the third term should be slightly larger since the cost of a comparison is proportional to the sums of the sizes of the objects concerned. However, the other object involved at this processor is always smaller than the larger object and so only a constant proportional error is introduced. Since we are looking for losses of efficiency which are functions of \( n \) or even \( S_0 \) this is not significant.) The first term is introduced by the observation that the real time between execution of each of the virtual processor’s instructions, is proportional to the number of other virtual processors simulated by the same real processor.

The time taken for iterations in which some of the real processors have become idle (and are therefore introducing gaps into the pipeline) will be estimated by various methods, justified as used. These can afford to be fairly crude since only inefficiencies of more than some constant factor are of interest.

**Example 1**

In this example it is assumed that the initial \( S_0 \) objects, all of size 1, are distributed evenly \( \frac{S_0}{n} \) per processor. In the first iteration these collapse to form \( \frac{S_0}{n} \) objects, each of size \( n \). For a further \( \log_2 \frac{S_0}{n} \) iterations, the number of objects is repeatedly halved in such a way that one increasingly large object is formed, while all other remaining objects are of size \( n \). In the worst distribution all the objects will be located in a single processor after the first iteration, with a gap of length \( n - 1 \) in the pipeline. In the best, the number of objects is kept well balanced between processors until only \( n \) remain. After this point, the number of active processors will be reduced by halves for the remaining \( \log_2 n \) iterations, creating gaps in the pipeline.
Chapter 5. The Iterative Combination Skeleton

The time for the first iteration is obviously common to both best and worst cases and is:

$$\frac{S_0 \cdot S_0 \cdot 1}{n} = \frac{S_0^2}{n}$$

There are a further \(\log_2 \frac{S_0}{n}\) iterations. In the best case the total time taken by all except the final \(\log_2 n - 1\) of these is

$$\frac{S_0}{n^2} + \frac{S_0}{n} + \frac{S_0}{2n^2} \cdot \frac{S_0}{2n} \cdot \left(\frac{S_0}{2} + n\right) + \frac{S_0}{4n^2} \cdot \frac{S_0}{4n} \cdot \left(\frac{3S_0}{4} + n\right) + ...$$

$$+ 1 \cdot n \cdot (S_0 - (n - 1))$$

$$= \frac{S_0^2}{n^2} \left(1 + \frac{1}{4} + \frac{1}{16} + ...\right) + \frac{S_0^3}{n^3} \left(\frac{1}{8} + \frac{3}{64} + ...\right)$$

$$= \Theta \left(\frac{S_0^3}{n^3}\right)$$

since only cases for which \(S_0 \gg n\) are of interest here. During the final \(\log_2 n - 1\) iterations the number of active processors falls by half between each iteration, thus introducing increasingly large numbers of gaps. A reasonable upper bound on their time will suffice. At each of these last iterations there are less than \(n\) objects all of size less than \(S_0\), and no more than one object resident in each processor. The time for each iteration is therefore \(O(nS_0)\) and this last phase takes \(O(nS_0 \log n)\). Thus,

$$Best\ case\ time = O \left(\frac{S_0^2}{n} + \frac{S_0^3}{n^3} + nS_0 \log n\right)$$

In the worst case, all objects are located in a single processor immediately after the first iteration. Time for subsequent iterations can be estimated by following the progress of the copy of the largest object present. Clearly this can encounter no holdups, since any bottlenecks in the virtual pipeline will occur immediately behind it. The time taken for the largest object has two components - the time spent in the real processor which simulates its home and the time spent being passed around the other empty processors. The first of these quantities is simply the product of the number of objects in the system, the number of objects with
which the largest object shares a real home processor and the size of the largest object. The second quantity is simply \( n - 1 \) times the size of the largest object. Thus, the worst case time after the first iteration is,

\[
\left( \left( \frac{S_0}{n} \right)^2 + n - 1 \right) \cdot n + \left( \left( \frac{S_0}{2n} \right)^2 + n - 1 \right) \cdot \left( \frac{S_0}{2} + n \right) + \\
\left( \left( \frac{S_0}{4n} \right)^2 + n - 1 \right) \cdot \left( \frac{3S_0}{4} + n \right) + \ldots + \left( 2 \cdot 2 \cdot \ldots + n - 1 \right) \cdot (S_0 - n)
\]

\[
= \frac{S_0^2}{n} \left( 1 + \frac{1}{4} + \frac{1}{16} + \ldots \right) + \frac{S_0^3}{n^2} \left( \frac{1}{8} + \frac{3}{64} + \ldots \right) + \\
(n - 1) S_0 \left( \frac{1}{2} + \frac{3}{4} + \ldots + \frac{n - 1}{n} \right) + n (n - 1) \log_2 \frac{S_0}{n}
\]

\[
= \Omega \left( \frac{S_0^2}{n} + \frac{S_0^3}{n^2} + n S_0 \log n + n^2 \log \frac{S_0}{n} \right)
\]

The main interest in comparing the best and worst case performance is in situations where \( S_0 \) is much larger than \( n \). For example, suppose that \( S_0 = \Omega (n^2) \). Then the best case time simplifies to \( O \left( \frac{S_0^3}{n^5} + n S_0 \log n \right) \), while the worst case simplifies to \( \Omega \left( \frac{S_0^2}{n^3} \right) \). If \( S_0 = \Theta (n^2) \) then the worst case performance is seen to be \( \Omega \left( \frac{n}{\log n} \right) \) times slower than the best case. Similarly, if \( S_0 = \Omega (n^3) \) then the times simplify to show that the worst case is fully \( \Omega (n) \) times slower.

To summarise, this example illustrates instances in which the number of objects falls dramatically in the first iteration then falls by halves at subsequent iterations. The combinations take place in a way which produces one increasingly large object together with other much smaller objects. For an initial set of objects sufficiently large with respect to the number of processors, it has been shown that performance of the implementation may vary by a factor as large as \( n \), the number of processors, if no redistribution technique is used.
Example 2

This example is similar to example 1. It differs in that the first dramatic iteration in which the number of objects fell from $S_0$ to $\frac{S_0}{2n}$, has been replaced by $\log_2 n$ iterations. The number of objects falls by half in each of these.

In the best case, successive iterations among the initial $\log_2 n$ will involve each processor having half as many objects as before. Thus the time taken in this initial phase is

$$
\frac{S_0}{n} \cdot S_0 \cdot 1 + \frac{S_0}{2n} \cdot \frac{S_0}{2} \cdot 2 + \frac{S_0}{4n} \cdot \frac{S_0}{4} \cdot 4 + \ldots + \frac{2S_0}{n^2} \cdot \frac{2S_0}{n} \cdot \frac{n}{2}
$$

$$
= \frac{S_0^2}{n} \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{2}{n}\right)
$$

$$
= O\left(\frac{S_0^2}{n}\right),
$$

which shows no significant change from example 1.

In a suitably bad case for our purposes (if not necessarily the worst) the new $\log_2 n$ initial iterations involve one processor losing no objects, while the others lose objects roughly by halves. Thus they will take time

$$
\frac{S_0}{n} \cdot S_0 \cdot 1 + \frac{S_0}{n} \cdot \frac{S_0}{2} \cdot 2 + \frac{S_0}{n} \cdot \frac{S_0}{4} \cdot 4 + \ldots + \frac{S_0}{n} \cdot \frac{2S_0}{n} \cdot \frac{n}{2}
$$

$$
= \Theta\left(\frac{S_0^2}{n} \log n\right)
$$

i.e. an increase of $\Theta (\log n)$ on the initial iteration from example 1. The rest of the execution proceeds as for example 1. Thus, the conclusions from example 1 still hold - an example has been presented which can exhibit a gap of $\Theta (n)$ in performance between best and worst distributions. More importantly, it can be seen that existence of the initial “dramatic” iteration of the previous example was not a necessary precondition to this result.
Example 3

The number of objects falls as in example 1 (including the "dramatic" first iteration). However the combinations occur more evenly - all objects gradually increase in size and no single object becomes much larger than all the others. Again both best and worst cases have the usual $\Theta \left( \frac{S_0^2}{n} \right)$ first iteration. Analysis of the subsequent iterations in the best case is split into two. The time for all but the final $\log_2 n - 1$ iterations (during which real processors are becoming idle) is

$$
\frac{S_0}{n^2} \cdot \frac{S_0}{n} \cdot n + \frac{S_0}{2n^2} \cdot \frac{S_0}{2n} \cdot 2n + \frac{S_0}{4n^2} \cdot \frac{S_0}{4n} \cdot 4n + \ldots + 1 \cdot n \cdot \frac{S_0}{n}
$$

$$
= \frac{S_0^2}{n^2} \left( 1 + \frac{1}{2} + \frac{1}{4} + \ldots \right)
$$

$$
= \Theta \left( \frac{S_0^2}{n^2} \right)
$$

In similar vein to example 1, it is noted that the final $\log_2 n - 1$ iterations involve objects of size varying between $\frac{2S_0}{n}$ and $S_0$, being passed around the full $n$ processor ring without bottlenecks. By the same argument, a bound of $O \left( nS_0 \log n \right)$ can be placed on the total time for these iterations giving an overall bound of $O \left( \frac{S_0^2}{n} + nS_0 \log n \right)$.

In the worst case all objects are located in a single processor after the first iteration. Following the argument of example 1 it can be seen that the time taken for the rest of the execution is

$$
\left( \frac{S_0}{n} \cdot \frac{S_0}{n} + n - 1 \right) \cdot n + \left( \frac{S_0}{2n} \cdot \frac{S_0}{2n} + n - 1 \right) \cdot 2n + \left( \frac{S_0}{4n} \cdot \frac{S_0}{4n} + n - 1 \right) \cdot 4n + \ldots + \left( 2 \cdot 2 \cdot n + n - 1 \right) \cdot \frac{S_0}{2}
$$

$$
= \frac{S_0^2}{n} \left( 1 + \frac{1}{2} + \frac{1}{4} + \ldots \right) + (n - 1) S_0 \left( \frac{1}{2} + \frac{1}{4} + \ldots + \frac{n}{S_0} \right)
$$

$$
= \Theta \left( \frac{S_0^2}{n} + nS_0 \right)
$$
Thus, for $S_0 = \Theta(n^2)$ the worst case is $\Omega(\log n)$ times slower while for $S_0 = \Omega(n^3)$ the discrepancy disappears. Most importantly, the dramatic variation seen in example 1 is no longer present, emphasising the important role played there by the large discrepancy in the sizes of objects.

Example 4

This example is like example 3, with the dramatic first iteration replaced by $\log_2 n$ iterations with the number of objects being halved in each. Again, no single large object is created. In the best case these iterations will take time

$$\frac{S_0}{n} \cdot S_0 \cdot 1 + \frac{S_0}{2n} \cdot \frac{S_0}{2} \cdot 2 + \frac{S_0}{4n} \cdot \frac{S_0}{4} \cdot 4 + \ldots + \frac{2S_0}{n^2} \cdot \frac{2S_0}{n} \cdot \frac{n}{2}$$

$$= \frac{S_0^2}{n} \left( 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{2}{n} \right)$$

$$= O\left(\frac{S_0^2}{n}\right)$$

and so do not affect the overall time for the phase significantly. In the worst case the initial iterations will take time

$$\frac{S_0}{n} \cdot S_0 \cdot 1 + \frac{S_0}{n} \cdot \frac{S_0}{2} \cdot 2 + \frac{S_0}{n} \cdot \frac{S_0}{4} \cdot 4 + \ldots + \frac{S_0}{n} \cdot \frac{2S_0}{n} \cdot \frac{n}{2}$$

$$= \Theta\left(\frac{S_0^2}{n} \log n\right),$$

an increase by a factor of $\Theta(\log n)$ over the dramatic first iteration and consequently in the total time taken. Considering cases for which $S_0 = \Omega(n^2)$, a gap of $\Theta(\log n)$ appears between best and worst case behaviour.

In Examples 1-4 it was assumed that initial set of objects was evenly distributed between the real processors. The next two examples give some insight into the significance of this assumption, by considering an alternative extreme, in which all objects are initially located in one processor. Consequently, the distinction between best and worst distributions no longer exists.
Example 5

All objects are located in the same processor throughout. The pattern of examples 1 and 2 is followed with respect to the appearance of a large object.

The time taken, assuming a dramatic first iteration, is

\[
(S_0 \cdot S_0 + n - 1) \cdot 1 + \left(\frac{S_0}{n} \cdot \frac{S_0}{n} + n - 1\right) \cdot n + \left(\frac{S_0}{2n} \cdot \frac{S_0}{2n} + n - 1\right) \cdot \left(\frac{S_0}{2} + n\right)
\]

\[
+ \left(\frac{S_0}{4n} \cdot \frac{S_0}{4n} + n - 1\right) \cdot \left(\frac{3S_0}{4} + n\right) + \ldots + (2 \cdot 2 \cdot + n - 1) \cdot (S_0 - 1)
\]

\[
= S_0^2 + n - 1 + \frac{S_0^2}{n} \left(1 + \frac{1}{4} + \frac{1}{16} + \ldots\right) + n (n - 1) \log \frac{S_0}{n} + \frac{S_0^3}{n^2} \left(\frac{1}{8} + \frac{3}{64} + \ldots\right) + (n - 1) S_0 \left(\frac{1}{2} + \frac{3}{4} + \ldots + \frac{S_0 - 1}{S_0}\right)
\]

\[
= \Theta \left(\frac{S_0^2}{n^2} + n^2 \log \frac{S_0}{n}\right),
\]

which is just \(\Theta \left(\frac{S_0^2}{n^2}\right)\) for the usual large \(S_0\). Similar analysis shows that this is not affected by the usual replacement of the dramatic first iteration by \(\log_2 n\) others.

This is the same as the worst case of example 1, in which the start was well distributed but immediately disrupted.

Example 6

As in example 5, all objects are permanently located in the same processor. This time the combinations are assumed to be more even, with no single large object appearing. With a dramatic first iteration the total time is

\[
(S_0 \cdot S_0 + n - 1) \cdot 1 + \left(\frac{S_0}{n} \cdot \frac{S_0}{n} + n - 1\right) \cdot n + \left(\frac{S_0}{2n} \cdot \frac{S_0}{2n} + n - 1\right) \cdot 2n +
\]

\[
\left(\frac{S_0}{4n} \cdot \frac{S_0}{4n} + n - 1\right) \cdot 4n + \ldots + (2 \cdot 2 \cdot + n - 1) \cdot \frac{S_0}{2}
\]
\[ S_0^2 + n - 1 + \frac{S_0^2}{n} \left( 1 + \frac{1}{2} + \frac{1}{4} + \ldots \right) + (n - 1) S_0 \left( \frac{1}{2} + \frac{1}{4} + \ldots \right) \]

\[ = \Theta \left( S_0^2 + \frac{S_0^2}{n} + nS_0 \right) \]

\[ = \Theta \left( S_0^2 \right) \]

for \( S_0 = \Omega \left( n^2 \right) \). Again, replacing the first dramatic iteration with \( \log_2 n \) halving iterations has no significant effect. The total time for suitably large \( S_0 \) can now be seen to be \( \Theta \left( n \right) \) times worse than that for the worst case of the same example with an even initial distribution.

**Example 7**

Examples 1-6 have all dealt with instances in which the number of objects was halved between iterations (sometimes with the exception of the first iteration). The final example considers an instance in which the number of objects present falls less rapidly. An evenly balanced start is assumed. The number of objects falls by \( n \) between each iteration, resulting in the creation of \( n \) increasingly large objects, and giving \( \frac{S_0}{n} \) iterations in all. As usual, best and worst cases refer to the distribution of other objects with respect to these. In the best case the \( n \) large objects are distributed one per processor. Each processor loses one small object at each iteration, until only the \( n \) large objects remain, to be combined in the final iteration.

In the worst case the embryonic large objects are all located in the same processor and will be built up there. This processor will lose no objects until all other processors are empty. Then it will lose its objects \( n \) at a time for each of the remaining iterations.
Chapter 5. The Iterative Combination Skeleton

The time in the best case is

\[
\frac{S_0}{n} \cdot S_0 \cdot 1 + \left( \frac{S_0}{n} - 1 \right) \cdot (S_0 - n) \cdot 2 + \left( \frac{S_0}{n} - 2 \right) \cdot (S_0 - 2n) \cdot 3 + \ldots + \\
+ 2 \cdot 2n \cdot \left( \frac{S_0}{n} - 1 \right) + 1 \cdot n \cdot \frac{S_0}{n}
\]

\[
= \sum_{i=1}^{S_0} \frac{i}{n} (S_0 - (i - 1)n)^2
\]

Using the facts [35] that

\[
\sum_{i=1}^{n} i = \frac{n(n-1)}{2}, \quad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6},
\]

and

\[
\sum_{i=1}^{n} i^3 = \left( \sum_{i=1}^{n} i \right)^2,
\]

it follows that

\[
\sum_{i=1}^{S_0} \frac{i}{n} (S_0 - (i - 1)n)^2
\]

\[
= \frac{1}{n} \sum_{i=1}^{S_0} i \left( S_0^2 + (i - 1)^2 n^2 - 2(i - 1) S_0 n \right)
\]

\[
= \frac{1}{n} \sum_{i=1}^{S_0} i S_0^2 + i^3 n^2 - 2i^2 n^2 + in^2 - 2i^2 S_0 n + 2i S_0 n
\]

\[
= n \sum_{i=1}^{S_0} i^3 - 2(n + S_0) \sum_{i=1}^{S_0} i^2 + \left( \frac{S_0^2}{n} + n + 2S_0 \right) \sum_{i=1}^{S_0} i
\]

\[
= n \left( \frac{1}{2} \frac{S_0}{n} \left( S_0 + 1 \right) \right)^2 - 2S_0 \left( \frac{1}{6} \frac{S_0^3}{n^3} \pm \text{lower order terms} \right)
\]

\[
+ \frac{S_0^2}{n} \left( \frac{S_0^3}{2n^2} \pm \text{lower order terms} \right)
\]

\[
= \frac{S_0^4}{6n^3} \pm \text{lower order terms}
\]

\[
= \Theta \left( \frac{S_0^4}{n^3} \right).
\]
Analysis of the worst case is divided into two phases. The first phase occurs while all the processors are still active. The second phase occupies the last $n^{th}$ of the iterations, during which the remaining objects are resident in a single processor and are simply passed on by all the others.

The first phase iterations in the worst case may be analysed in similar fashion to the best case, with a simple alteration to reflect the fact that one of the processors is losing no objects. Thus it takes time

$$\sum_{i=1}^{n-1} \frac{i S_0}{n} (S_0 - (i - 1) n)$$

Similar manipulations lead to the result that the time for the first phase of the worst case is also $\Theta \left( \frac{S_0^2}{n^8} \right)$. Analysis of the second phase of the worst case reverts to the techniques employed in examples 5 and 6 to deal with situations when all processors but one are empty, i.e. by using $(|S|^2 + n - 1) \cdot |\text{largest object}|$ as a measure of the time taken by an iteration.

By this method, the time taken for the second phase in the worst case is seen to be

$$\sum_{i=1}^{S_0} \left( \left( \frac{S_0}{n} - (i - 1) n \right)^2 + n - 1 \right) \left( \frac{n - 1}{n^2} S_0 + i \right)$$

which simplifies to $\Theta \left( \frac{S_0^2}{n^8} \right)$. This is not significant in comparison with the time taken by the first phase in both cases. We can therefore conclude that

$$\text{total best case time} = \Theta (\text{total worst case time})$$

for this example, and consequently that any attempts at redistribution are unlikely to be worthwhile.

### 5.4.4 Summary and Discussion of Examples

Any conclusions drawn from the preceding analyses must be prefaced by two qualifications, noting the inevitably small number of examples and the dangers
of placing too much emphasis on $\Theta()$ type comparisons (especially when these appear to show "no significant difference" between quantities). Nevertheless, it would be taking such scepticism too far to dismiss completely the emergence of certain trends. These have a bearing on implementation decisions relating to the distribution of objects.

In this spirit, the following points should be noted:

- a well balanced initial distribution of objects is essential (consider examples 5 and 6, and compare with examples 1 and 2);

- without the appearance of large objects redistribution cannot offer dramatic savings (consider examples 3 and 4);

- the slower the rate of appearance of large objects, the less effect they have (compare example 7 with examples 1 and 2), presumably since more work is done in the early stages when objects are still reasonably distributed;

- the most important factor in redistribution is the balancing of the number of objects per processor, independent of size (this is a direct result of the simple implementation method proposed).

To summarise, the analysed examples suggest that a good initial distribution is essential and that the value of dynamic redistribution increases with both the likelihood and speed of the appearance of a significant imbalance in the sizes of remaining objects.

In practice, information about the procedure for merging objects is available before execution, and it would be possible to use this to decide whether to employ redistribution on a problem by problem basis. However, such decisions would
only be meaningful in the context of a particular implementation on a particular machine.

5.4.5 Implementing Redistribution

The examples presented above demonstrate the existence of problem instances for which the absence of some redistribution technique allows the possibility of a \( \Theta(n) \) fold deterioration in performance. The remaining question is whether such a scheme can be implemented at justifiable cost, with respect to the time saved. For the present depth of analysis this appears to be possible, using the scheme now presented.

At the end of each iteration, processors execute the sum and broadcast algorithm to ascertain the total number of objects remaining (this is already necessary for termination checking). From this information, each processor decides whether it has a local surplus or shortage of objects. Processors having a surplus send unwanted packets off on another tour of the physical ring. These will be grabbed by processors with a shortage. All processors can determine that the phase has finished by putting an extra "marker" packet on the ring after any surplus objects dispatched. When this packet returns (it is simply passed round the ring), the phase is known to be complete. This is essential since none of the surplus objects will themselves return.

Although this appears a rather costly exercise (which it is), the crucial observation is that the time taken will be no larger than that spent dealing with the next iteration. In fact, in most cases it will be somewhat smaller. Therefore, it

---

\(^3\)In this respect, the whole problem is reminiscent of that of caching policy in memory system design and the *ad hoc*, but undoubtedly effective, decisions made there.
Chapter 5. The Iterative Combination Skeleton

will only have the effect of increasing the total time by a factor of \( \leq 2 \) (independent of \( n \) and \( |S| \)). This is obviously justifiable when considered in the light of the potentially \( \Theta(n) \) fold saving produced.

The result is a clear illustration of an important point in the ubiquitous computation-communication trade-off. The fact that a large amount of time has been spent doing essential work efficiently (i.e. the "real" work of each iteration), allows a generous amount of time to spent on invisible (in terms of the abstract skeleton specification) house-keeping, without dramatically affecting overall performance or scalability.

5.5 Redistribution with a Shortage of Objects

The issue of redistribution can appear in a complementary way to that discussed above. Instead of a surplus, consider the situation which occurs when the number of remaining objects falls below the number of processors. Clearly an increasingly large number of processors will become idle and, if the implementation continues to use the full tour of the grid, the time wasted simply passing the few remaining objects between idle processors may become significant. Thus, some form of redistribution of objects would appear to be called for. Ideally, remaining objects would be assigned to home processors after each iteration of the "test, select, combine" phase, in such a way that all new homes are connected by a Hamiltonian circuit of length equal to the number of remaining objects. An easy way to ensure this would be, whenever possible, to select a suitably sized square sub-grid of processors as the new homes. Equally, it is important that the cost of performing this relocation is not so large as to nullify any associated gain in performance. It would appear sensible to implement redistribution only when the number of objects has fallen substantially with respect to the number of
processors still involved in the tour. Accordingly, an algorithm is now presented which implements redistribution once the number of objects has fallen to half the number of active processors.

5.5.1 The Goal

The crucial iterations are those in which, as a result of combinations, the number of remaining objects has fallen to half (or less) the number of processors responsible for its implementation. In general, suppose that the processors implementing the iteration occupied a $k \times k$ sub-grid, where $k$ is guaranteed\(^4\) to be a power of 2, and that the number of remaining objects (calculated during the termination checking phase) has fallen to be no more than $\frac{k}{2^i} \times \frac{k}{2^i}$, for some integer $i$, but more than $\frac{k}{2^{i+1}} \times \frac{k}{2^{i+1}}$. In such situations, the goal of the redistribution algorithm is to relocate the remaining objects to new homes occupying some sub-grid of processors of dimension $\frac{k}{2^i} \times \frac{k}{2^i}$.

Arbitrarily, the sub-grid at the centre of the original is chosen, as illustrated in figure 5–5. Some of the remaining objects will already be located at homes in this inner sub-grid, where they may remain. Others will currently have homes in the outer belt of processors, which are all now idle. These "free" objects must be relocated to processors within the inner grid currently having no home object. Such processors will be referred to as "holes". It should be noted that the number of holes is as least as large as the number of free objects, by virtue of the size of the selected sub-grid. A processor may immediately determine whether it has a free object or is a hole as a direct consequence of its absolute location in the

\(^4\) Note that this is not a serious restriction. It simply requires that the original hardware be manufactured with the number of processors being a power of 2. It places no restrictions on the use of the skeleton.
grid and the total number of objects remaining. The task of the redistribution algorithm is to move each free object to a unique hole as quickly as possible. The algorithm presented below achieves this goal within time guaranteed to be $O(k \log k)$. This comes close to matching the trivial lower bound on worst case performance obtained by considering an instance with one free object and one hole, which may be located $k$ links apart, even by the shortest route.

5.5.2 A Suitable Algorithm

The algorithm presented here is more general than is actually required, in that it will allow holes to be located anywhere in the original $k \times k$ grid. The situation
here, with holes restricted to the selected $\frac{k}{2^i} \times \frac{k}{2^i}$ sub-grid is just a particular case. Any choice of location for the new sub-grid would be handled equally well.

The core of the approach taken is to repeatedly partition the $k \times k$ grid into smaller and smaller pieces, while correcting the distribution of free objects between each division as appropriate to the number of holes therein. Partitioning of each area is always into two halves, with any two successive partitions being orthogonal. Thus, the first level of partition splits the $k \times k$ grid into two halves and balances free objects between them. The second level concurrently splits each half in two, producing two pairs of $\frac{k}{2} \times \frac{k}{2}$ square sub-grids. The crucial observation is that it is now only necessary to balance within each pair, since balance between pairs has been ensured at the previous level. Redistribution within each pair proceeds entirely independently and concurrently. As the level of partition increases, so does the number of such concurrent activities and their locality. At the $\log_2 k^2$th level of partition the redistribution takes place between adjacent pairs of processors, thus completing overall redistribution satisfactorily.

It remains to describe the process by which some group of processors is partitioned and the free objects present are balanced between the two halves.

Consider the $i^{th}$ level of partition. An initial group of $\frac{k^2}{2^{2i-1}}$ processors is divided\(^5\) into two groups of $\frac{k^2}{2^i}$. Suppose that these contain $f_1$ and $f_2$ free objects and $h_1$ and $h_2$ holes respectively. Then, since $f_1 + f_2 \leq h_1 + h_2$ is guaranteed (otherwise the algorithm would not have been called), it may be the case that either $f_1 > h_1$ or $f_2 > h_2$ but certainly not both (i.e. at most one half may contain more free objects than holes). All that is required of the redistribution algorithm is that it should detect and rectify this situation at each level of partition.

---

\(^5\)By partitioning a square into two equal rectangles or a rectangle into two equal squares, as $i$ is odd or even respectively.
Suppose that the original area was rectangular and that the partition therefore produces two squares. The algorithm proceeds in three phases, as follows:

1. Concurrently, in each square, processors run a localised copy of the sum and broadcast algorithm of section 4.2.1 to determine the number of free objects, \( f_* \) and free holes, \( h_* \) in their square. Comparison of these allows each square to determine whether it should be exporting free objects.

2. Processors in an exporting square determine precisely which \( f_* - h_* \) objects will be sent across the boundary. Arbitrarily, these are chosen to be those residing in processors of lowest identifier, assuming a row major ordering. To achieve this, processors in each row cooperate to count the total number of free objects in the row. Then, processors in the first row subtract their row total from \( (f_* - h_*) \) before passing the result to their neighbour in the row below. If the result is positive then all processors in the row having free objects must send them across the partition. If the result is negative, then only as many objects as would have made the result zero must be dispatched. Again, these are chosen by smallest home processor identifier. Such processors can identify themselves by sending a count along the row from lowest to highest identifier. The process ripples down the rows until, upon completion in the bottom row, those processors which will send their free objects across the boundary have been identified.

3. The selected free objects are shunted across the boundary, in the manner of heap redistribution of section 4.3.3. Thus an object originating in the \( j^{th} \) row and \( j^{th} \) column of the exporting square moves to the corresponding processor in the importing square. Of course, this receiving processor is not guaranteed to be a hole, and may already be responsible for an object. At the next level of partition it may be required to shunt both of these objects
elsewhere (more locally). In general, at the \(i^{th}\) level of partition a processor may become responsible for as many as \(2^i\) objects. Thus the shunting at the next level will be executed as a sequence of \(2^i\) single object (for each processor) shunts.

At this point, it is certain that \(f_1 \leq h_1\) and \(f_2 \leq h_2\) and that more localised redistribution can continue independently in each square, with concurrent instantiations of the algorithm just described. Cases where the original grid is square and the partition produces two rectangles require only a trivial adaption of the sum and broadcast algorithm. We now consider the execution time of the whole redistribution operation.

**Theorem 13** The entire algorithm executes in \(\Theta (k \log k)\) time.

**Proof:** There are \(\log_2 k^2 = 2 \log_2 k\) levels of partition. It is now shown that the operations at each level are performed in \(\Theta (k)\) time, thus producing the required result.

The first phase involves the simple sum and broadcast on a square (or rectangular) grid of side length no more than \(k\), and hence \(\Theta (k)\) time. The second phase involves a rippling of information down columns of length no more than \(k\), with only constant delay between rows, and possibly one ripple along a similarly sized row. This again takes \(\Theta (k)\) time. Finally, the third phase involves the shunting of objects between adjacent sub-grids. As noted above, there are \(2^{i-1}\) such shunts at the \(i^{th}\) level. However, the decreasing size of the sub-grids involved means that each such shunt is only of distance (and hence time) \(\frac{k}{2^{i-1}}\), if \(i\) is even and \(\frac{k}{2}\) if \(i\) is odd. Thus, the total time taken for the phase is just \(\Theta (k)\), the product of these, irrespective of level. Overall, each phase, and therefore the whole operation, takes \(\Theta (k)\) time at every level, and the whole \(2 \log_2 k\) level algorithm
Chapter 5. The Iterative Combination Skeleton

takes $\Theta(k \log k)$ time.

5.5.3 Evaluating the Algorithm

The examples presented in section 5.4.3, dealing with the performance of the proposed redistribution algorithm for $|S| \gg n$, illustrated the difficulties involved in trying to gain a useful insight into such a general problem. Analysis of the algorithm for $|S| \leq n$ is further complicated by the fact that the sizes of the "initial" objects can no longer reasonably be assumed to be equal. These objects may actually be inherited from previous iterations (i.e. from an initial $|S_0| \gg n$), and so may be of arbitrarily large and varied size with respect to $n$. Consequently, it appears that any analyses of examples along the lines used previously would carry very little weight. At this stage, the only safe conclusion is that a reasonable decision on whether to implement the $|S| \leq n$ redistribution scheme could only be taken in the context of a particular machine, on the basis of experience with real examples.

5.6 Alternative Approaches

A possible implementation of the iterative combination skeleton has been presented. The techniques employed were shown to tackle their respective problems in a straightforward, efficient manner. Nevertheless it would be rash to claim that these necessarily represent either the only or the best such methods. In this section some other possibilities and their inherent difficulties are briefly considered.

The most obvious criticism which can be made of the proposed testing phase is that it calculates the cost of merging every possible pair of remaining objects,
even those for which neither member has been altered from the previous iteration. There are clearly instances in which such repeated work will constitute a significant proportion of all the work done during the phase. An alternative approach might try to ensure that only pairs involving at least one new object are tested. Only details of new objects would need to be routed to other processors, and this could possibly be performed in a more direct way than the simple, case-independent tour. However, there are two obvious drawbacks to such a system. In the first place it would require processors to store details of all previous tests and all old objects. Results of previous tests would have to be maintained in some ordered fashion, and both sets of data would have to be carefully updated to preserve correctness. This would involve substantial cost in both space and time at each processor. Secondly, if the rigid routing scheme of the tour was to be abandoned in favour of something more case sensitive, then the well known problems of contention for links and bottlenecks would have to be considered, especially for cases in which most objects were actually new and had to be distributed globally.

Similar arguments apply to the merging phase, in which selected objects are actually combined. For example, it would be possible to route objects more directly to their partners. Once again the usual congestion problems would appear, compounded by the activity introduced by any grouping of object combinations.

A further problem shared by such alternative schemes concerns the detection of the end of the phase. In the tour implementation this is implied by the return of a processor's own object (in the test phase) or marker (in the merge phase). Processors in the alternative schemes would presumably have to co-operate in some kind of global polling to detect completion. This would allow the method to be general enough to quickly complete iterations requiring little work, while still catering for longer examples. Such a mechanism could involve substantial work, especially in cases where several such polls were initiated before the end of the phase actually occurred.
These considerations suggest that while no all embracing proof can be offered as to the superiority of the proposed method, its combination of simplicity and reasonable efficiency make it a good pragmatic choice.

5.7 Examples

5.7.1 Minimum Spanning Tree and Connected Components

Sollin’s algorithm [8] to find a minimum weight spanning tree of a weighted graph was presented in section 5.1.1 as an introduction to the iterative combination skeleton. The details required to adapt the skeleton to this algorithm for a \( v \) vertex graph would be

- a type definition of a tree, including its component vertices and edges and the set of edges leading from component to non-component vertices. Each edge should include a note of its weight,

- a function which combines two such trees by their shortest joining edge and returns a description of the resulting tree together with a note of the weight of the joining edge. If no such edge exists then the function can return an empty tree and an “infinitely” high weight,

- a function which compares the costs returned by two possible combinations and accepts the lower of the two (or neither if both are “infinitely” large),

- a collection of \( v \) trees, one for each vertex of the graph, consisting of the vertex identifier, an empty set of component edges and a set containing a description of each edge adjacent to the vertex.
Execution of the skeleton could terminate in two ways. If there is only one object remaining then this will contain a description of a minimum weight spanning tree of the original graph. Alternatively, execution may terminate with a collection of objects remaining, between which no edges exist. In these cases, the original graph is not connected (and consequently has no spanning tree). However, the objects remaining each represent a maximal connected component of the original graph. Thus, a similar implementation can also be used (if somewhat inefficiently) specifically to locate the connected components of a given graph.

Now consider the efficiency with which the skeleton will execute the MST algorithm given a $v$ vertex graph. A straightforward implementation might represent a tree by a length $v$ array of records, with the $i^{th}$ entry noting the length and end-points of the shortest edge which joins vertex $i$ to any vertex present in the tree (this will be 0 if $i$ is already in the tree and $\infty$ if $i$ is not adjacent to the tree), and a linked list of edges which make up the MST of the vertices present in the tree.

The function to combine two such trees creates the shortest edge array of the combined tree by setting its $i^{th}$ entry to the minimum of the $i^{th}$ entries of the two existing arrays. While constructing this, it keeps note of the smallest difference found between any two $i^{th}$ entries, where one was zero. This corresponds to the shortest edge between the two trees. If this is finite, then the process is completed by linking together the two lists of component edges and adding the newly found edge, to represent the edge list of the new tree. If the shortest edge was of "infinite" length, then the trees are not directly adjacent and cannot be combined at this stage. The whole operation will take $\Theta(v)$ time in all cases. The function to compare two combinations selects that of lower cost. This requires only constant time.
Considering the example in which the number of trees is halved at each iteration, we see that the sequential implementation will take time
\[ v \left( v^2 + \left( \frac{v}{2} \right)^2 + ... + 2^2 \right) = \Theta \left( v^3 \right). \]

Analysis of the parallel implementation follows a pattern similar to that presented in example 4 of section 5.4.3. For the MST example it must be noted that the size of "objects" is now \( \Theta \left( v \right) \) throughout (because of the array). The overall time taken is therefore \( \Theta \left( \frac{v^3}{n} \right) \) for \( v = \Omega \left( n^2 \right) \). Thus the skeleton implements the algorithm very efficiently with respect to the sequential method described.

However, it is not difficult to see that the sequential method can be substantially improved. For example, if the description of a tree is augmented to include a list of member vertices, then the shortest edge leaving each tree can be found by inspecting only against these vertices. The run time of the sequential algorithm is then reduced to \( \Theta \left( v^2 \right) \).

Unfortunately, this improvement has no effect on the skeleton implementation. It is still necessary to move objects of size \( \Theta \left( v \right) \) around the ring and the overall time is unchanged, making the parallel implementation highly inefficient! This is because the new sequential algorithm no longer makes use of the full capacity presented by the skeleton specification. The cost of combining two objects is now found without actually testing the result of the combination and therefore without having to combine the objects at all. The efficiency of the skeleton is based on performing these test combinations in parallel (and relies on them to hide its communication overheads). Therefore, it is not surprising that in making them redundant, the efficiency of the skeleton is destroyed.

The lesson to be drawn from this example is that skeletons should only be adapted to examples which exploit the full problem solving power presented in the abstract specification.
5.7.2 Partitioning Programmable Logic Arrays

The programmable logic array (PLA) [45] is one of the most commonly used building blocks in the design of VLSI systems. A PLA provides a simple, automatable means of implementing a collection of combinational functions which may share common inputs and minterms. The main problem associated with their very regular structure is that without careful (and time consuming) design large internal areas may be effectively wasted, thus using up valuable chip area. The partitioning approach attempts to overcome this by dividing the initial, crude PLA into several smaller ones which collectively implement the same functions but which require a smaller total area. Unfortunately, the problem of finding a guaranteed optimal partition is too computationally demanding to be feasible in practice. Consequently, heuristic approaches to the problem have been considered. One of these [13,25] proceeds in a manner which closely resembles the iterative combination skeleton.

The original PLA is divided up into a large collection of smaller PLAs (a typical choice is to have one for each original minterm). These are then combined on the strength of a cost function which calculates the area required by a PLA. A combination is acceptable if it requires less area than its components (a reduction in area arises from the sharing of common inputs and minterms). Eventually the process will stop with either a collection of PLAs representing a successful partition of the original, or a single PLA (exactly the original) indicating that no partition was found (but not necessarily that none exists).

In terms of the skeleton, the details required are

- a type definition of a PLA including details of inputs, outputs and minterms,
- a combine function which can merge two such descriptions (taking possible sharing into account) and return the resultant decrease (or increase) in area,
• a compare function which compares two such costs and accepts the larger, provided that this is positive, and

• descriptions of the initial PLAs, constructed by dividing up the original.

A typical implementation represents a PLA as a boolean array indexed by input, product and output nodes, where elements indicate the presence or absence of nodes. The size of such an object is therefore $v$, the total number of nodes in the original PLA, and the cost of merging two of them (by OR-ing together arrays) is $\Theta(v)$ independent of their internal details. Realistic PLAs tend to have $|\text{products}| = \Theta(|\text{inputs}| + |\text{outputs}|)$ and so the initial problem size is also $\Theta(v)$.

From these assumptions, the analysis is identical to that of the simplistic MST algorithm of the previous example, resulting in $\Theta(n)$ fold speed-up for $n$ processors, when the problem is suitably large. Most importantly, there is now no short-cut available to the sequential algorithm at the expense of the skeleton. Any improvement in representation would apply equally to both implementations. The important point is that it is now strictly necessary to examine the result of a combination of two PLAs in order to calculate the resultant saving. Consequently, the communication time in the skeleton implementation cannot be exposed as it was previously. In summary, PLA partitioning provides an example of an application which can exploit the iterative combination skeleton effectively.
Chapter 6

The Cluster Skeleton

6.1 An Alternative Approach to Skeleton Construction

The skeletons discussed in previous chapters share a similar pattern of development. Each was initially motivated by the isolation of a particular algorithmic technique, apparently possessing some scope for parallel execution. Having tied down the abstract specification of the skeleton, it only remained to describe an implementation which could exploit the inherent parallelism in the context of the realistic machine. Skeletons were designed for the convenience of the user, rather than that of the implementor, responsible for maintaining a reasonable level of efficiency.

The skeleton presented in this chapter is the result of an experiment in the reversal of this approach. Here, we begin with an attempt to quantify some interesting operation for which the structure of the grid seems especially suited. Then, from the bottom up as it were, we construct an abstract specification with its implementation based on this operation.
6.2 Motivating a New Skeleton

The most obvious way to use a grid of processors efficiently is to handle problems whose structures directly match the neighbour-neighbour communication pattern presented by the machine. Much real computing time is spent doing just this. However, the observation that “two dimensional grids are good at simulating two dimensional grids” hardly seems likely to be the source of an interesting skeleton.

Instead, the construction of the new skeleton begins with another simple observation about the structure of the square grid, inherited from its more general role as a rectangular grid. We note that rectangular grids are highly amenable to partitioning into sets of smaller, mutually independent rectangular grids. Furthermore, for any particular initial grid, there is a very large number of such partitions, exhibiting a wide variation in the relative sizes of the sub-grids produced.

Our interest in this property is strengthened by the observation [9] that it doesn’t hold for the constant degree graphs with logarithmic diameter often discussed in the context of parallel computation, such as the degree 2 de-Bruijn graphs (the “2-shuffle” [10]). Essentially, while the edges in the logarithmic diameter networks are distributed to diverge quickly (in order to produce the low diameter), those in the grid are more localised.

Building out from this we note that Theorem 11 is easily adapted to show that any rectangular grid with at least one side of even length contains a Hamiltonian circuit. Thus, the square grid is well suited for partitioning into independent sub-grids, each containing a Hamiltonian circuit. Of course, the sub-grids themselves will share this property. The remainder of this chapter discusses the specification and grid implementation of a skeleton which can exploit this property.
6.2.1 Building a Skeleton

We have noted that the grid is suitable for partitioning into more or less arbitrarily sized pieces, and that this can be done in such a way as to ensure that each contains a Hamiltonian circuit.

Turning to the issue of problem specification, these properties echo important features of two of the skeletons discussed previously. Specification of the recursive divide and conquer skeleton centred on the partitioning of a problem into similar sub-problems. Meanwhile, the iterative combination skeleton made much use of a Hamiltonian circuit of the grid and in particular its suitability for the “pipelining” of operations. We now expand upon these observations to specify a new skeleton.

The recursive divide and conquer skeleton required its problems to be split into some specified number of sub-problems of equal size. This division of problems and the subsequent combination of results was performed centrally. The sub-problems themselves were distributed for independent, concurrent solution. The unrestricted nature of the division of the grid now under consideration, suggests that the requirement that sub-problems should be of equal size is no longer relevant. Equally, the co-operative style of solution by the use of Hamiltonian circuits seems to point away from the centralised division and combination found in the recursive divide and conquer skeleton towards a more distributed implementation of these functions. This suggests that the description of problems should follow the more disjoint style of the iterative combination skeleton.

Taken together, these considerations point towards the notion of “clustering” together of component parts of whole problem into groups whose independent solution contributes in some way to the global solution. We may wish to deal with each cluster in a different way depending upon the property describing each. Crystallising these ideas, we propose that the new skeleton will deal with problems for which data sets of instances can be described as a collection of homogeneous
objects whose individual descriptions may include information which relates them to each other. This is identical to the type of specification required by the iterative combination skeleton.

Solutions to such problems will proceed by dividing the objects into independent subsets and recursively dividing these subsets as often as is possible (or suitable). The process of sub-division will be referred to as “clustering” since it will involve partitioning sets into subsets representing clusters of the original objects where all members of such a cluster share some property or are “close” in some sense. The clustering process imposes a hierarchical structure of clusters onto the set of objects with the original complete set at the root.

As the recursion unwinds, members of clusters are considered together with all other members of their parent cluster and operated upon with respect to these. In this way each object is manipulated as a member of a succession of increasingly general clusters. As with the iterative combination skeleton the procedures used to divide and recombine clusters will be designed to take advantage of the Hamiltonian circuits of processors present in the underlying machine. Thus, to divide a cluster of objects into a set of sub-clusters each object will be compared with each other. A sub-cluster will be constructed for every maximal subset of objects which are connected directly or transitively by the specified notion of “closeness” (i.e. two objects will appear in the same sub-cluster if they are “close” or are connected by a sequence of “close” objects). The measure or property used to judge this may be parameterised by the depth of the current cluster in the hierarchy. Similarly, in recombining clusters, all pairs of objects present will be considered and manipulated appropriately. Again, the procedure used may vary in some way with the cluster’s depth in the hierarchy, or with some other characteristic.
6.2.2 The Abstract Specification

The preceding discussion has produced a loose description of a new skeleton arising from consideration of a particular property of the grid. It is now appropriate to specify the user's view of the skeleton precisely.

The user is required to provide a type definition of the objects which will comprise the data set of a problem instance, together with a set \( S \) of such objects describing the particular instance in question. The whole evaluation can then be considered as consisting of a call "cluster\((S, 0)\)" of a procedure "cluster" as specified in figure 6–1. The user is required to provide a boolean function "continue" which decides whether to continue with clustering on the basis of the

```plaintext
PROCEDURE cluster (S : object set; i : integer);
VAR S' : set of object sets;
BEGIN
  IF continue(S,i+1) THEN BEGIN
    decompose(S, S', i+1);
    FOR each set \( s \) in \( S' \) DO cluster\((s, i+1)\);
    compose(S',S, i+1);
  END;
  Handle_cluster(S,i);
END;

Figure 6–1:
```

depth reached and possibly some property of the set (eg. its size). Any such property must be decidable by examining any member object.
The procedure "decompose" takes a cluster $S$ and returns a set of clusters $S'$. The method used to form the clusters will depend upon $i$, the depth in the hierarchy of cluster $S$. The procedure is as specified in figure 6–2. The clustering

PROCEDURE decompose ($S$ : object set; $S'$ : set of object sets; $i$ : integer);
BEGIN
  FOR each object $s$ in $S$ DO
    FOR each other object $t$ in $S$ DO
      IF suitable($s$, $t$, $i$) THEN include($s$, $t$, $i$);
  END;
END;

Figure 6–2:

process has similarities to that used for merge selection in the iterative combination skeleton, but is more general. Here, each object may collect a set of others which it requires to be among those sharing its cluster at the next level. These sets are coalesced to form the actual clusters which are the members of $S'$. In specifying the method to be used to form clusters the user must provide:

- a type definition of the data structure to be used to store the set of required partners for an object\(^1\); 

- a boolean function "suitable" which decides whether an object $t$ should be included in the partner set of an object $s$ at level $i$. The decision can be

\(^1\)Since each instance of this structure will always be handled by a single processor, it may take any form allowed by the selected language.
made on the basis of the descriptions of s and t and the current contents of the partner set of s;

- a procedure "include" which describes how the partner set of s should be adjusted to include t at level i. For example, this might involve adding to the end of a list, inserting it according to some order of precedence or simply replacing the whole set with the single object t.

The procedure "compose" simply recombines a set of clusters into a single cluster as specified in figure 6–3.

PROCEDURE compose (S' : set of object sets; S : object set;
   i : integer);
BEGIN
   FOR each set s in S' DO
      FOR each object t in s DO add t to S;
   END;

Figure 6–3:

Finally, the procedure Handle_cluster (described in figure 6–4) requires the user to provide the procedure "modify", which adjusts the description of object s to take into account the presence of object t in the same cluster at depth i. For example it may be required to implement significant interaction between objects clustered together for large i, with less interaction at higher levels.
PROCEDURE Handle_cluster (S : object set; i : integer);
BEGIN
  FOR each s in S DO
    FOR each other t in S DO modify(s,t,i);
  END;
END;

Figure 6-4:

6.3 Implementing the Skeleton

In this section we discuss a possible grid implementation of the cluster skeleton based on the properties considered in the introduction. We begin by giving an outline of the whole procedure before investigating the components in more detail.

6.3.1 Structure of the Implementation

The specification of the skeleton is founded upon the use of recursion to build up a hierarchy of clusters of objects. This principle is reflected in the proposed implementation. Each processor is associated with a particular cluster at every level of the hierarchy in such a way that its cluster at some particular level is a sub-cluster of its cluster at the next higher level. This allows operations on objects at lower levels of the grid to be kept entirely independent of activity involving unrelated clusters. At a particular level \( l \) a processor will co-operate with the other processors sharing the same cluster to implement the decomposition into sub-clusters. It will then push details of its neighbours at that level onto a local
stack before sharing responsibility for one of these at the next level of recursion. Upon returning from the recursive call of "cluster" it pops the information from the stack and co-operates with its former partners at level $l$ to implement the procedures "compose" and "Handle_cluster".

Using a grid partitioning of the type discussed in the introduction we can ensure that at each level processors involved with the same cluster are always connected by a circuit, and can therefore implement the "all possible pairs" comparisons efficiently.

### 6.3.2 Dealing With One Level

We now consider the actions required of a group of processors to deal with a cluster $S$ at some level $i$ of the hierarchy. We can assume that each processor in the group already knows the number of objects in the cluster and the number of processors in the group. We can also assume that:

- the descriptions of the objects in $S$ are distributed evenly (or as nearly as possible) between the processors in the group;

- there exists a circuit of physical links connecting the processors;

- each processor has copies of all the user defined type definitions and procedures.

These conditions are obviously satisfiable at the level 0, when the "cluster" contains all the objects and the "group" encompasses the whole grid. Section 6.3.3 will show how they may be maintained from one level to the next.

Execution begins with evaluation of the condition "continue($S, i + 1$)". Since we have already required that it should be decidable upon examination of any
Chapter 6. The Cluster Skeleton

member of S it can be evaluated independently and concurrently by all processors in the group, with each being guaranteed to reach the same conclusion.

If the decision is to continue, the processors move on to execute the decomposition of the cluster. Each virtual processor creates and maintains the "partner set" for its object, using the specified type definition. Processors co-operate to implement the nested FOR loops of the procedure "decompose" in identical fashion to the all-pairs merge testing of the iterative combination skeleton. Each virtual processor sends a copy of its object around the circuit. On receipt of such a copy \( t \), a virtual processor executes the function "suitable(\( s, t, i \))" and, if necessary, the procedure "include(\( s, t, i \))" using its resident object \( s \). As before, instances in which processors are responsible for a number of objects are dealt with by having each real processor simulate several virtual processors. The loop terminates upon the return of each object to its own processor.

Continuing the analogy with the iterative combination skeleton, each processor now knows the identifiers of at least some of the objects with which its own object will share a cluster at the next level. The next step is to calculate the identifier of that cluster. Adopting the convention that clusters share the lowest identifier of their members it is clear that we can borrow directly the implementation of Savage's connected components algorithm from section 5.3.2 to allow this information to be gathered. Here, each member of each partner set contributes an edge to the pool. This completes execution of the procedure "decompose" for cluster \( S \) at level \( i \).

Processors in the group now execute the algorithm discussed in section 6.3.3 which re-organizes the grid in preparation for the recursive call of "cluster". On return from the call, each processor resumes its role as a member of the group at level \( i \). As a result of activities at the lower levels the processor may now be responsible for objects different from those which it held before recursion.
However, the operations introduced in section 6.3.3 ensure that these are still members of the same cluster. Thus, procedure "compose" is implemented at no cost by simply forgetting about the lower level clusters from which the objects were returned.

It only remains to perform the work associated with procedure "Handle_cluster". Again, this involves passing copies of objects around the circuit at level $i$, with each processor executing procedure "modify($s, t, i$)" for its own object $s$ and every other object in the cluster $t$, as it receives them.

6.3.3 Moving Between Levels

In this section we discuss one method by which a group of processors dealing with a cluster at some level of the hierarchy can reorganize themselves to deal with the subclusters formed at the next level of recursion. By emphasising locality (in the sense of grouping by Hamiltonian Circuits) we ensure that these activities and associated transfers may be performed entirely independently of other groups of processors dealing with other clusters. Thus, the details apply equally to moves between any two levels. The method is initially presented in a slightly idealised form. The minor additional difficulties which must be overcome for implementation on the grid are then considered.

An Idealised Solution

We have a group of $p$ processors which are responsible for a cluster of $c$ objects at some level of the hierarchy. The descriptions of the objects are distributed evenly between the processors and all include a note of the identifier of the sub-cluster to which they will belong at the next level down. The processors are physically connected by a Hamiltonian Circuit of links. The goal is to re-assign objects to
processors within the group such that these conditions are retained by all sub-clusters and associated groups of processors at the next level. Furthermore, this must be done in such a way that the relative sizes of the sub-clusters are reflected by the relative sizes of the groups of processors handling them.

In the idealised solution we make two extra assumptions, namely that \( p \) is even and that the processors may be numbered from 1 to \( p \) such that \( P_i \) is always physically adjacent to \( P_{(i-1) \mod p} \), \( P_{(i+1) \mod p} \), (both by virtue of the links which form the circuit) and \( P_{-i+1} \). These additional links are shown as vertical dotted lines in figure 6–5 and two processors so joined will be referred to as a "pair". This section will show how the first assumption can be satisfied, while the subsequent section on "practicalities" will deal with the second.

Suppose that the \( c \) objects are to be partitioned into \( k \) sub-clusters of sizes \( c_1, c_2, \ldots, c_k \). Then the best possible reorganisation would assign these to groups of \( \lceil \frac{c}{e} \cdot p \), \( \lceil \frac{c_2}{e} \cdot p \), \( \ldots \), \( \lceil \frac{c_k}{e} \cdot p \) processors respectively (to the nearest integer). Our solution will approximate this by assigning to groups of \( d_1, d_2, \ldots, d_k \) processors, where \( d_i \) is the largest even number no larger than \( \frac{c_i}{e} \cdot p \) when \( \frac{c_i}{e} \cdot p \) is at least 2, and 1 when \( \frac{c_i}{e} \cdot p \) is less than 2. It has three phases, each of which is completed in \( \Theta (c + p) \) time. This is small enough to be hidden in an asymptotic analysis by the similar \( \Theta (c + p) \) time spent on the "real" work of the decompose procedure.

In the first phase, processors gather information about the sizes of clusters to
be re-assigned. For every object, one copy of the identifier of its new cluster is passed around the circuit. By inspecting these, each processor can accumulate the sizes of all the clusters.

In the second phase processors calculate the re-assignment of clusters. With the exception of those only entitled to a single processor (i.e. those for which $\xi p < 2$) clusters are assigned to groups of processors from left to right along figure 6-5, in increasing order of cluster identifier.

Consider an example in which $c = 70$ and $p = 20$. Suppose that the first two clusters are of size $c_1 = 22$ and $c_2 = 13$. Then the approximation requires that these be allocated to groups of 6 and 2 processors respectively, as illustrated in figure 6–6.

This procedure is followed to allocate all clusters entitled to two or more processors, before the remaining clusters (each entitled to only one processor) are assigned to the remaining processors. Since we have tended to underallocate processors until this point, we would usually expect at least as many processors to remain as there are unassigned single processor clusters. In such a situation we can simply allocate remaining clusters to distinct processors in increasing order of both cluster and processor identifier (as occurs in the completion of the example). As a special case, we must be prepared for a situation in which there are more single processor clusters than free processors. The simple solution
is to allocate more than one cluster per processor, making individual physical processors simulate the operations of several virtual processors. 2

To complete the example, suppose that the remaining clusters are of sizes $c_3 = 1, c_4 = 33, c_5 = 1$. The resulting complete allocation is illustrated in figure 6–7.

![Figure 6–7: The complete allocation](image)

There are two important points to note about the allocation procedure. Firstly, it is completely deterministic for given $c, c_1, \ldots, c_k$ and $p$. Consequently, given the accumulated information about the sizes and identifiers of clusters, each processor can independently ascertain the identifier of the cluster for which it will share responsibility at the next level and its position within the appropriate group. This is achievable in $O(c)$ time since it involves a sequence of at most $c$ unit cost arithmetic operations and checks.

Secondly, the allocation ensures that each cluster is shared between a group of processors connected by a circuit of physical links. Thus clusters at the next level of the example will be handled by the circuits illustrated in figure 6–8, allowing the whole procedure to be repeated at the next level.

---

2This is entirely analogous with the situation occurring in the recursive divide and conquer skeleton when the physical leaves are required to simulate the operation of several virtual levels.
Chapter 6. The Cluster Skeleton

174

Figure 6-8: Circuits at the next level

It only remains to move the descriptions of objects to processors in the appropriate groups. Since each processor now knows the identifier and size of its cluster at the next level it can easily calculate the number of objects for which it will be responsible. Once again, passing one copy of each object description around the higher level tour allows each processor to collect the right number of new objects (the choice between particular processors in the group for particular objects within a cluster is irrelevant).

This completes the procedure for moving between activities at successive levels of the cluster hierarchy. All processors now push onto a stack a note of their circuit neighbours at the old level before proceeding with operations at the new level (in some cases with different neighbours produced by the sub-division of the tour). On completion of the lower level, the old neighbour descriptions are popped from the stack and the code associated with "Handle_cluster" at the old level is executed.

Practicalities

In the abstract solution it was assumed that the links shown as dotted lines in figure 6–5 were present as direct physical connections, in addition to those forming the circuit. It is not difficult to see that there is no mapping of identifiers to square grid processors which satisfies this condition except for the trivial $p = 1$. 
and \( p = 4 \). However, the general mapping technique illustrated in figure 6–9 comes close enough to make a simple adaption of the abstract solution practical. The regularity of the abstract circuit is broken at positions where the real circuit turns a corner. At these points the processor on the “outside” of the bend has no obvious partner, while that on the inside has two possible partners (e.g. consider the top right-hand processor). This has the effect of introducing “unsuitable” breaks into the re-allocation process, these being points at which allocation of one group of processors may not be terminated. Figure 6–10 illustrates such a situation. It is unsuitable to break the circuit portion of part (a) as illustrated in

![Figure 6–9: A useful Hamiltonian circuit](image)

![Figure 6–10: An illegal break](image)
and therefore wasted\(^3\). Instead, the allocation must move one pair back as shown in part (c), with the next allocation beginning satisfactorily.

The revised, practical algorithm must take such points into account and will therefore allocate the greatest even number of processors which is no larger than that to which the cluster is entitled, and which does not introduce an unsuitable break. Fortunately, the locations of unsuitable break points remain static throughout computation (being a structural property of the original full tour, independent of the level of hierarchy) and this additional consideration introduces no technical problems and only constant additional time.

6.4 Examples & Discussion

6.4.1 Multi-Key Sorting

We now present a multi-key sorting algorithm which can be implemented by use of the cluster skeleton. Suppose that we have a set of objects which must be sorted on the basis of \(k\) keys indexed \(1..k\) in decreasing order of significance, and that no two objects are identical with respect to all keys. Each object has a "position" field which will eventually note its position in the sorted set.

An obvious algorithm to solve this problem can be based on a multi-level "bucket sort". Objects are partitioned into buckets by examination of their most significant key field, with objects of equal key value being placed in the same

\(^3\)Although the loss of several processors may be acceptable at the end of allocation, it is not possible to allow wastage during allocation since this may result in a shortage later on.
bucket. Buckets are repeatedly sub-divided by examination of decreasingly significant key fields. Then, moving back up through the bucket hierarchy each object increments its "position" field (initially set to 1) by one for every other object in the same bucket having a smaller value in the key field by which that bucket was previously partitioned. Eventually the position field will have been incremented by one for every object which precedes that object in the required ordering.

In terms of the cluster skeleton this involves

- a type definition of the objects, including their key fields and a "position" field,

- a set of such objects, each with "position" set to 1,

- a boolean function "continue(i)" with result set to "i<k",

- a data structure capable of representing a set of object identifiers,

- a boolean function "suitable(s,t,i)" with result set to s.key[i] = t.key[i],

- a procedure "include(s,t,i)" which adds the identifier of object t to the partner set of object s,

- a procedure "modify(s,t,i)" which executes

  if s.key[i+1] > t.key[i+1] then
  s.position := s.position + 1;

After execution the position field of each object will note that object's position in the sorted set.
6.4.2 Image Analysis

As a second example, consider a problem which may occur when analysing some body of data describing some scene or event. For example these may be digitised images of portions of some overall picture. Typically, there may be a large number of such observations, originating from a variety of sources and possibly "blurred" in some way. In order to make sense of such a mass of information, it may be sensible to ascertain (or at least to speculate) which observations actually refer to the same object and to coalesce each such group into a clearer and hopefully more accurate description, thus distilling a coherent picture of the scene.

A detailed description of the type of techniques used in comparing and coalescing such images is not of interest here (see [29] for a recent discussion). It is sufficient to note that these are of varying degrees of complexity and consequently, execution time. In deciding which observations refer to the same entity it would be clumsy to apply the most complex test to every pair, when a much simpler procedure may be able to discriminate between most. Thus, we can envisage a sequence of tests producing a hierarchy of observation clusters, and the parallel with the cluster skeleton is clear.

At the top level, the initial set contains representations of all the observations. This is partitioned by some quick, simple test, powerful enough to distinguish between those which are obviously disparate. At subsequent levels, increasingly subtle (but time consuming) tests are applied to observation clusters of decreasing cardinality. At the lowest level, each remaining cluster is assumed (on the basis of the tests used) to contain objects representing the same "real world" entity. In order to obtain what is hopefully a more accurate representation of the entity, the observations are coalesced in some appropriate way. Returning similarly through the hierarchy, we emerge at the top level with a coherent (and hopefully more accurate) representation of the observed scene, with multiple observations of the
same entity now matching in terms of description but maintaining their own source location information and so on.

6.4.3 Discussion

In similar style to the iterative combination skeleton, the implementation of the cluster skeleton derives its efficiency from phases involving the repetition of some operation across all pairs of objects from some set. Given \( n \) such objects, the \( n^2 \) possible pairings are examined \( p \) at a time by \( p \leq n \) processors, thus achieving optimal speed up for these operations. The work required to re-assign objects to new groups of processors is sufficiently concise to affect only the constant in the asymptotic execution time.

The two examples presented here emphasize an important point about the use of skeletons in general. Consider an instance of the sorting example in which \( n \) items are being sorted by \( n \) processors, and assume that the time required to compare any pair of keys is constant. It is not difficult to see that the parallel implementation will reduce execution time by a factor of \( \Theta(n) \), with the precise details depending upon the distribution of objects to clusters. Essentially, the sequential execution time of \( \Theta(kn^2) \) is reduced to \( \Theta(kn) \) in parallel. This certainly demonstrates the efficiency of the parallel implementation, but hides the fact that a better choice of algorithm could reduce the sequential time to \( \Theta(kn \log n) \). Not surprisingly, the parallel implementation cannot remedy an inappropriate choice of sequential algorithm. In general, if the selected skeleton is an inefficient choice in its sequential form then the performance improvement afforded by parallel execution is only evident with respect to the weak starting point.

In contrast, the second example describes a good choice of skeleton. Here, all the work performed in the sequential specification (and hence in the parallel implementation) is necessary for the solution of the problem as described. In
this case, the improvements introduced by the parallel hardware are genuine. We noted a similar situation in the examples presented in chapter 5.
Chapter 7

Conclusions

As the computational power afforded by highly parallel computers becomes indispensable to its users, so the problem of building useful and usable systems to harness it becomes more pressing. The designer of such a system is required to find an acceptable balance between two conflicting objectives. On the one hand, users of the system require it to present some abstraction which allows clear and easy specification of solutions to their problems. In particular, the abstract system should be independent of the details of the underlying parallel hardware, and the historic problem of non-portability caused by the inclusion of machine specific characteristics at higher levels must not be repeated. On the other hand, the absolute performance of the system will be expected to match that obtainable (at least in the manufacturer’s imagination) by grappling directly with the problems of communication, synchronisation and distributed memory. “Why did I spend $x$ on ten thousand processors if they only solve my problems a hundred times faster?” cries the irate user. The fact that identical performance would be deemed quite acceptable if the black box purchased for the same price contained only one hundred processors is overlooked.

In chapter 1, we considered the existing spectrum of systems which address the problem. This led, in chapter 2, to the proposal of a new style of abstraction,
Chapter 7. Conclusions

based on the notion of algorithmic skeletons. Subsequent chapters centred on the specification and implementation of one such system, comprised of four skeletons, on a particular model of parallel hardware. By way of conclusion, this chapter presents a discussion on the merits of the “skeletal machine” with respect to existing systems. Finally, some possible paths for the future evolution of the new approach are mapped out.

7.1 The Case for Skeletons

The concept of the skeletal machine has been proposed as an alternative to those discussed at the beginning of the thesis. We now attempt to throw some light upon its merits and failings in the context of these systems, concentrating on general principles rather than the minutiae of particular implementations.

As was noted in chapter 2, the key feature dividing the “skeletal machine” from all the others is the fragmented nature of its high level abstraction. Whereas all the systems discussed in chapter 1 present some general purpose programming language, possibly in conjunction with a machine model, skeletons are entirely independent of each other. In effect, each defines its own special purpose “machine” designed to execute some particular style of algorithm. This divergence from the mainstream of existing work has both strengths and weaknesses. By their very existence, skeletons serve to highlight “good” algorithmic style at a high level, encouraging the user to describe solutions well suited to parallel implementation and, perhaps, suggesting previously unconsidered approaches. The system may be likened to a helpful advisor, repeatedly asking “Can you visualise a solution to your problem looking like this, or this, or this ...”. In contrast, existing systems at the higher levels of abstraction (those discussed in section 1.2.1) present some language suitable for solving any problem, but give no suggestion as to how to
construct specific solutions. At the other end of the spectrum, systems with a low level of abstraction based on explicit fixed networks seem to say "If your problem looks like a grid (or whatever) then you've cracked it, otherwise too bad"! Of course, the choice of multiple specialisation over complete generality introduces its own problems. In particular, what happens when the "helpful advisor" runs out of ideas? In short, the answer is that the user must abandon the skeletal abstraction and try some other package. However, the range of examples presented here suggest that this will not occur frequently. Furthermore, it must be emphasised that the four skeletons presented are in no way assumed to be exhaustive. Future studies of existing algorithms and attempts to use the abstraction by a wide base of users would no doubt result in the construction of a much wider range of skeletons.

In this context, the independence of the various skeletons with respect to one another is very important. New skeletons can be added without affecting the behaviour of those already present. Eventually, it is possible to envisage a situation in which the absence of a suitable skeleton for some problem strongly suggests that the problem itself is unsuitable for efficient parallel evaluation.

At the abstract specification level, the major issue dividing existing systems concerns the degree to which explicit parallelism is allowed or enforced. The purer versions of the highly abstract machines of section 1.2.1 prohibit any reference to it. While this has the undoubted benefit of sheltering users from the less pleasant aspects of parallel programming, it also precludes the possibility of expressing a range of solutions most naturally described in terms of simple concurrency. In contrast, the systems discussed in sections 1.2.2 and 1.2.3 force the user to adopt explicit parallelism whether suitable or not.

In this area the skeletal approach has a definite advantage. For the most part (e.g. three of the four skeletons presented here) skeletons represent purely
sequential abstractions from the realms of conventional computing. However, it is easy to include explicitly parallel skeletons, where these seem relevant, without extending the parallelism across the whole abstract machine. The user can just as easily be guided towards good parallel solutions as towards sequential solutions with good parallel implementations. Independence between skeletons allows these styles to co-exist safely and conveniently within the same overall abstract machine.

In common with the systems of sections 1.2.1 and 1.2.2 the abstract specification of the skeletal machine is pitched at a level which makes it entirely independent of the hardware. Thus, complete portability of "programs" (i.e. the user specified procedures required to customise skeletons) is ensured from one underlying architecture to another. The only requirement is that the sequential language used to specify such procedures be catered for on the processors comprising the physical machine. This is a purely sequential problem. Thus, from the user's viewpoint, only performance may vary from machine to machine, with the logical behaviour of programs guaranteed to be consistent.

Continuing with the subject of implementation on a variety of architectures, it seems likely that the fragmented nature of the skeletal machine will again prove advantageous. The implementation of a monolithic programming language must handle a large range of possible eventualities and program structures. In producing such a system, many concessions must be made in terms of efficiency in order to guarantee universality. Furthermore, the flexibility of such abstractions permits the specification of programs having no efficient parallel implementation.

In contrast, the independence of the individual skeletons breaks the implementation task into several more restricted parts, each dealing with only a particular pattern of execution. The system designer may concentrate on implementing each in isolation, without having to consider a full range of awkward possibilities. Similarly, the user is prohibited from describing completely unsuitable computations.
7.2 Future Directions

The whole notion of an abstract machine based on algorithmic skeletons is clearly in its infancy. Looking to the future, there are two obvious directions in which to diversify. At the abstract level there is scope for a wider selection of skeletons than those presented here. As we have seen, these may be extracted either from existing sequential algorithms or from the rapidly expanding field of explicitly parallel algorithms. In this area, the contributions of experienced practitioners should be invaluable. An obvious example is "dynamic programming". The systolic algorithm presented in [23] seems to present a suitable foundation for such a skeleton.

Similarly, at the implementation level, it seems desirable to extend the range of hardware upon which the skeletal abstractions can be executed. The new model's independence of any particular architecture is an important feature which should be exploited. In chapter 2, the square grid of processor-memory pairs was selected as a particular model of parallel hardware upon which to consider implementation of the system. The primary concern in making the choice was that the hardware chosen should be unarguably realistic and that asymptotic results concerning the efficiency of implementation be reflected in practice, with acceptable constant factors. These conditions are certainly satisfied by the grid. In progressing from paper studies to actual implementation it would be foolish to be needlessly bound by restrictions more applicable in theory than practice. For example, the emerging popularity of the hypercube family as a practical communications network linking substantial numbers of processors should not be dismissed casually. To ignore a network capable of efficiently connecting \(2^8\) (for example) elements on the grounds that it will struggle with \(2^{16}\) (or whatever) seems over cautious in practice. For
such a machine we would hope and expect to see the ubiquitous $\Theta(\sqrt{n})$ time factor improved to $\Theta(\log n)$.

It would be quite possible to advance independently in these two directions without producing any inconsistency. However, this "ad hoc" direct implementation of each skeleton from scratch on each new architecture would result in much wasted effort. The possibility of creating some standard level of interface between abstract specification and implementation levels (imitating the role of the Alice compiler target language [54] in the design of the Alice machine) appears inviting.

A suitable format for such a level becomes apparent upon consideration of the four implementations presented previously. Here, a set of underlying operations and patterns of data manipulation emerge. For example, the ability to efficiently simulate the behaviour of complete trees of processes (for a variety of fixed degrees) was of central importance in chapter 3, while the operations of sorting and routing across the network were much in evidence in chapter 4. Global and local coalescence and broadcasting of information appeared in several contexts. The implementations presented in chapters 5 and 6 were founded upon the existence of Hamiltonian circuits linking all processors and the suitability of the network for partition into conveniently related sub-graphs was prevalent in chapters 4 and 6.

The construction of a new intermediate level with a specification providing such structures and operations would provide a convenient rendezvous between implementor and abstract skeleton designer. New skeletons would be described in terms of facilities provided at the intermediate level. Similarly, implementations on new hardware would only address the problems involved in providing them, rather than dealing directly with the skeletons. Of course, as with the concept of skeletons itself, there would be no barriers to the extension of the intermediate level, either from above or below.

It was noted in chapter 2 that the introduction of a new level of abstraction
to a system tends to pay its price in terms of efficiency. The success or failure of such an intermediate level in the skeletal machine would obviously depend upon the suitability of the facilities provided. An inappropriate or incomplete intermediate level could become more of an obstacle than an aid. On the other hand, a good choice would free those involved in designing the higher levels from architectural considerations, while providing a less esoteric target for the "low-level" architectural experts. In this way, the wide range of issues which required consideration in this thesis would be conveniently divided.
Bibliography


Bibliography


Bibliography


