Algorithmic Skeletons: Structured Management of Parallel Computation

Murray I. Cole

Department of Computing Science,

University of Glasgow,

United Kingdom.

[Now of the Division of Informatics, University of Edinburgh. The text is reproduced on the WWW by kind permission of the publishers, MIT Press.]

Table of Contents

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 efficiently 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 book 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, in the way in which "higher order functions" represent general computational frameworks in the context of functional programming languages. 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 "fixed degree divide and conquer", "iterative combination" "clustering" and "task queues". 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, and 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

This book is a revised and extended version of my Ph.D. thesis. Many thanks are due to Gordon Brebner who supervised the original work at the University of Edinburgh. He provided timely encouragement and advice, 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 was supported financially at Edinburgh by a Science and Engineering Research Council research studentship.

The material on skeletons as higher order functions has been developed since my arrival at the University of Glasgow. Particular thanks here are due to John Hughes for pointing out the connection between the two concepts.

Finally, I would like to thank Tom Lake for his careful and constructive observations during the revision process.

This book is dedicated to Ally, who knows why.

Murray Cole,
Glasgow,
1989.

Chapter 1

Generating and Controlling Parallelism

1.1 Introduction

The task of designing and implementing any part of a computer system is essentially 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.

Each abstraction allows us to sacrifice 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 exploiting 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 abstraction process can be seen to make a wider range of solutions accessible in practice (though not, of course, in theory). Furthermore, these higher level solutions reap significant gains in portability and clarity.

The inevitable decline in performance associated with abstraction has an important corollary to the effect 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 a functional 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 many 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, 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 by miniaturisation.

As an idea this is nothing new. In the very first issue of the Communications of the ACM, Saul Gorn [11] 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."*

Whereas 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 lower levels 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? These are the questions underpinning the work presented in this book.

We are concerned with the process of designing and implementing high level programming systems which can exploit massively parallel hardware. Note that the word "systems" is used deliberately here in preference to "languages". We are more interested in the abstract computational model implied by a particular language than with a precise syntax. Depending upon the approach taken, the amount of real language design involved in the process can vary from none at all to the complete specification of a new language. We will see examples of both and of instances falling between these extremes in the material presented in the remainder of this chapter. The new approach proposed subsequently will be seen to fall towards towards the former end of the scale. It allows the bulk of a program to be described in an existing language, adding just enough superstructure to significantly ease the task of parallel implementation. In order to put these proposals in context, we begin with a review of existing approaches.

1.2 The Spectrum of Existing Systems

A variety of techniques are currently being used to address the problem of building higher level programming systems upon parallel hardware. When less important details are filtered out, the resulting systems fall into three rough categories.

The level of abstraction in the first category is high. Users of these 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 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 contains systems in which the user is required to specify solutions which use parallelism in a style very close to that physically present.

The use of a parallel computer to tackle some problem involves encounters with several well known sources of difficulty. The designer of a parallel programming system must decide which of these to handle implicitly and which to leave to the programmer. In the ensuing discussion we will consider the choices made in the design of existing systems. It will become clear that these essential problems are inextricably interwoven and that it makes little sense to consider any one in isolation. However, in order to set the scene and clarify terminology, we will briefly present the main characteristics of each.

The most fundamental and obvious task is that of *problem decomposition*, the identification of parallelism. Processes must be described which can operate concurrently to achieve a solution, or some indication of how these can be generated dynamically must be found. We should be certain (or at least very confident, in less predictable circumstances) that the parallel solution will be faster than a traditional sequential solution! Unfortunately, the eventual performance will be governed by a host of other implementation issues but an optimistic starting point is clearly essential.

The second problem is that of *distribution*, the physical exploitation of the potential parallelism identified by decomposition. We must specify a mapping from operations which may be executed concurrently to the available processors, or as before, indicate the mechanism by which this mapping can be achieved dynamically.

It is most unlikely that decomposition and distribution of a problem will lead to a situation in which a large number of processes perform entirely independent operations upon entirely independent data. More realistically, our parallel processes will perform sequences of similar or identical tasks upon data which is shared, to a greater or lesser extent. The third problem to be addressed is that of implementing this *code and data sharing.* The obvious choice is between the replication of information (with the associated costs in space and consistency maintenance) and direct sharing (with the problems of granting and controlling access efficiently).

Finally, it is necessary to consider the precise mechanisms by which the sharing of resources (whether code or data) is to be described and controlled. *Communication* and *synchronization* underlie the whole notion of concurrent operation and will be present in at least one and probably many levels of any parallel system.

We are now in a position to examine the relevant characteristics of our three categories more closely. In doing so, we inevitably omit many other fascinating aspects of the systems involved. References [29, 4, 2, 10] provide good starting points for further reading.

1.2.1 Highly Abstract Systems

The first category contains systems in which the abstract machine presented to the user is entirely devoid of parallelism and is completely isolated from the underlying implementation mechanism. Such systems typically present functional, logical or data-flow models of computation. They are often referred to as being "declarative" systems, since the programmer makes a series of declarations which define the properties of a solution to some problem, rather than specifying a precise series of operations which will lead to the solution. Thus, languages of this type (at least in their pure forms) are not only non-parallel but also non-sequential, having no notion at all of a flow of control.

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 evaluating the result of applying a top-level function to an argument. This type of function application is the only operation present and involves the replacement of a function-argument pair with a copy of the function body (from its definition) in which occurrences of the "dummy" or "free" variable have been replaced by copies of the actual argument (which may of course itself be a function application). Functions and arguments may be nested. This simple system can be shown to provide as much computational power as any other fundamental computing mechanism (e.g. the Turing machine). A particularly powerful aspect of the model is the ability to define "higher order functions", to which we will return in chapter 2. Other convenient features such as multiple argument functions, localised definitions and data structures may all be defined as lambda expressions.

In the same way, a high level functional program is simply a 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 employing the function definitions (conceptually using the application by substitution technique from the lambda calculus) to obtain an alternative, but equivalent representation of the function and arguments pair. This "output" is simply a more useful representation of the original program and "input", in the way that "6" is a more useful representation of "$(4-2)*3$".

The important point is that execution may progress from the initial to the final representation in any fashion which maintains equivalence. In particular, it will often be possible to execute many 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). An obvious way to represent the program as it evolves is as a graph, in which nodes represent function applications. The children of a node are the arguments of the corresponding application. The process of expanding and contracting the graph (i.e. program evaluation) is referred to as "graph reduction".

With this approach, the task of decomposition to generate parallelism is simple. The abstract execution model allows candidate nodes to be expanded at any time, while function applications may be evaluated as soon as arguments are available. Thus, a potentially parallel process is generated every time a node reaches one of these states.

It is important to realise that this does not imply that every functional program is actually highly parallel. As a trivial example, consider defining a function to compute factorials. The obvious definition will look something like this:

$$\begin{aligned} \mathit{factorial}\ 0 &= 1 \\ \mathit{factorial}\ n &= n * \mathit{factorial}\ (n-1) \end{aligned}$$

Such a function would execute sequentially on a typical graph reduction machine, irrespective of the number of available processors. A more complex definition notes that

$$\begin{aligned} \mathit{factorial}\ 0 &= 1 \\ \mathit{factorial}\ n &= \mathit{product}\ 1\ n \\ \mathit{product}\ a\ a &= a \\ \mathit{product}\ a\ b &= (\mathit{product}\ a\ \lfloor \tfrac{a+b}{2} \rfloor) * (\mathit{product}(\lfloor \tfrac{a+b}{2} \rfloor + 1)\ b) \end{aligned}$$

which produces significant potential parallelism. Although declarative systems involve no explicit notion of execution sequence, it is unfortunately clear that some knowledge of the execution mechanism can be used to great effect by the programmer.

The main problem for the implementor of a functional system comes with the realistic distribution of the available parallelism. In graph reduction, the structure of the graphs produced is specific to each problem instance. Furthermore, this structure only becomes apparent during execution and evolves dynamically. Thus any mapping scheme which tries to distribute the graph and the associated workload effectively must be both dynamic and general purpose. This problem can be tackled in two ways. The first approach [16] 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 [7] 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. There is a difficult 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.

An alternative approach [15] recognizes the difficulty of automating distribution and allows the introduction of program annotations which provide a means of influencing the execution mechanism. These are guaranteed to preserve the semantics of the computation, but may improve its efficiency. Such additions may be argued to move the model out of this category, in that the programmer is now partly (or even wholly) responsible for the task of decomposition. Similarly, [20] discusses a language which allows program partitioning and interconnection structure to be described in a declarative style.

The distinction between "data" and "code" in a functional system is blurred – both are intermingled in the graph during expansion, reflecting their common foundations in the 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. Most systems [7, 16] 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 concurrently.

A functional program contains no explicit notions of communication or synchronization. However, in a realistic implementation these are introduced as a by-product of decomposition, distribution and sharing, and must be handled by the system itself (and consequently by the implementor of that system).

The data flow model of computation [9] arrives at a similar point to graph reduction by a different route. 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 basic control mechanism is that a node may only perform its operation once it has received data tokens on all of its inputs. Thus, nodes may execute in parallel, subject only to the availability of data. A typical data-flow graph will be re-entrant and for any realistic problem, there will probably be more operator nodes in the graph than there are 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, a data flow graph is statically defined by the program and only data is manipulated. Languages built upon these concepts are still "functional" but may be dressed up to resemble (at least superficially) sequential imperative languages [3], particularly if "scientific" applications are envisaged. The compilation process from high level language to the underlying data-flow graph has some similarity to the process of expansion in graph reduction and amounts to the decomposition phase of parallel implementation.

There is nothing analogous to the function definitions of functional language schemes, since these have been compiled away. 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. Thus, a good distribution scheme is difficult to obtain. Existing solutions follow the two patterns observed in the functional case in choosing whether or not to attempt to localise operations upon certain portions of the graph to particular processors.

Finally, logic languages 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. In similar fashion to functional programming, the specification of a computation consists 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. More interestingly, it is 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.

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. As with graph reduction, the semantics of pure logic languages often allow this process to proceed at many points in parallel. Similarities between the two styles are emphasised in [8].

1.2.2 Idealised Parallel Systems

The systems discussed in the first category were characterised by the isolation of the abstract (or idealised) design space seen by the programmer 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 programmer's world includes explicit parallelism. The programmer is now responsible for decomposing 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.

In the "shared memory" camp, much theoretical work on parallel algorithms focuses on the use of a variety of classes of idealised parallel machines (e.g. see [10]). Such machines consist of a collection of n standard uni-processors each with its own local memory. 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. Concurrent-read exclusive-write (CREW) machines allow an arbitrary number of read accesses to the same location but require writing to be unique, while CRCW models 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 augment the instruction set to include operations such as "fetch and add". Concurrent execution of several such operations on some location l produces the same result as if the instructions had been executed in some unpredictable sequential order. Thus, each participating processor receiving a copy of the original contents of l incremented by a unique subset of the other "added" values, while l itself has its contents incremented by the sum of all the values specified in the instructions.

Proposed implementations of such machines associate each 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 idealised step.

Straightforward solutions map each 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 idealised 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 [26] to generate the shared to local memory mapping and of randomising routing phases [37] intended to alleviate the problems associated with awkward message permutations. In spite of good average or expected behaviour, any such scheme is vulnerable to 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 an n fold simulation slow down, a performance level which could be matched by a single processor[1].

A proposed remedy to this problem [26] is to keep multiple copies of each SM location in different real processor memories and to access only the most convenient. Initially this was proved successful only for dealing with multiple

[1]This is true even of EREW models, since it is not necessary that the n requests refer to the same location but merely to the same processor's local memory.

reads. However, in a striking paper Upfal and Wigderson [36] show that the technique can be efficiently extended to include concurrent write steps. Several practical projects, such as the New York University "Ultracomputer" [12] have drawn from this pool of theoretical ideas.

The occam language [18] [2] 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 decomposition of data into independent sets in addition to the decomposition 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.

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 system to user. No support is provided for problem decomposition, with the programmer explicitly 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 typically provide facilities for explicit inter-process communication. The burden here is still carried by the system which provides abstractions to more general networks than are physically present in hardware.

Finally, a variety of approaches are evident to the issue of synchronisation.

[2] occam is a trademark of the INMOS group of companies.

At one extreme, the straightforward progression through a shared sequential program provided in several theoretical models 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 relatively simple. The more practical shared memory models are the least helpful here, requiring careful use of shared variables to ensure process rendezvous, though instruction sets augmented by "fetch and add" style instructions are useful.

1.2.3 Low Level & Restrictive Systems

The final category completes the journey through the levels of abstraction afforded by parallel systems. It includes systems 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 [25] is a good 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, but 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.

Turning to the key issues, it is clear that systems such as this represent a completion of the transition from system to user responsibility. The programmer is entirely responsible for problem decomposition and distribution. Code sharing will be implemented by simple replication, while any data sharing and communication required must be programmed explicitly with direct reference to the physical machine topology. Some support for the mechanics of synchronisation may be provided (e.g. by machines which run occam code on each node) but it will be up to the programmer to determine how and when to use this.

1.3 Conclusions

The preceding section categorised approaches to the problem of providing a useful, general purpose programming model combined with an implementation efficiently harnessing the computational power of parallel hardware.

The highly abstract approach of the first category has 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. [22]) report on only relatively small scale machines.

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.

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.

In summary, existing systems either present a relatively friendly abstraction with hidden, unpredictable 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 essentially without overheads.

While varying widely in the degree of machine independence provided, all of these approaches share one common principle. In each case, the programming model presented to the user is intended to be "universal" in the sense that its full range of mechanisms may be used in the specification of any computation. In the more abstract machines this freedom has the result of allowing the description of computations which will be difficulty to implement in parallel. In the low level machines, geared directly towards parallel implementation it produces systems which are hard to use in many situations. The work presented in the subsequently proposes and investigates a new approach to the problem, based on the rejection of the concept of a single, universal programming model.

Chapter 2

Algorithmic Skeletons – A New Approach

In this chapter we introduce the "algorithmic skeleton" as the concept which will underpin our proposed approach to the provision of high level programming systems for parallel machines. We begin by discussing the lessons which can be learned from the survey introduced in chapter 1 and consider their implications. Section 2.2 describes the overall structure of the new approach.

In subsequent chapters we will present specifications and possible implementations of four candidate skeletons. Consequently, section 2.3 introduces the model of parallel hardware upon which the implementations will be built, while section 2.4 discusses the methods which will be employed in their construction and the measures which will be used to predict their efficiency.

2.1 Motivation

In the first chapter we considered a variety of programming systems which shared the goal of facilitating the efficient use of parallel hardware. Our investigation focussed on the issues of problem decomposition, distribution, code and data sharing, communication and synchronisation. We examined the ways in which particular systems divide responsibility for these between implementation and programmer. In retrospect, the resulting categorisation may be seen to be closely linked to the way in which systems handle the first two issues. Thus, in the first category full responsibility for both is allocated to the implementation, the second category leaves decomposition to the user but handles distribution in the implementation, while the third category makes the programmer deal with both. This neat correspondence should not be surprising, since decomposition and distribution are clearly the two most fundamental issues involved. While solutions to the other problems are important and will have serious implications for performance, the type of sharing and communication required is determined by the way in which decomposition and distribution are handled. Without a parallel algorithm there can be no messages to exchange or data to share.

More importantly, the survey also illuminated one common principle which unites all categories. This is the notion that the model of computation and its associated programming constructs should be "universal" in the sense that they may be employed unrestrictedly in solving any problem. The motivating thesis for our new approach is that this "universality" is at the root of the serious difficulties encountered by existing approaches when dealing with problem decomposition and distribution. Thus, at the more abstract end of the spectrum, the expressiveness of the languages allows the programmer to specify computations which are either inherently sequential or so obscurely parallel that any realistic implementation will struggle to find a good decomposition and distribution strategy. As the simple "factorial" example demonstrates, this can be true even if the problem itself does admit an implicitly parallel solution. At the other end of the spectrum, systems targeted towards specific hardware have such restrictive models of computation (e.g. "anything that looks like a hypercube") that the programmer may struggle to find good solutions to problems with anything but the most natural and obvious decomposition. Between these extremes, the difficulties are more equitably shared between system and programmer, but must still be overcome.

Following from this analysis of the situation, we are tempted to conclude that the provision of a single, universal programming framework is incompatible with the goal of uniformly efficient parallel implementation. If the latter property is to remain a target then we must discard the former. However, our rejection of "universality" should not also throw out the high level of abstraction and machine independence which we associate with good systems. Indeed, our target system should not even be explicitly parallel, since we have already noted that problem decomposition is a difficult task for the programmer. Our goal then, is to define a programming system which, while appearing non-parallel to the programmer, admits only programs which it can guarantee to implement efficiently in parallel.

The solution which we propose draws its inspiration from two sources. Firstly, we note that the challenge of designing good algorithms in a purely sequential context has been the subject of extensive study. As experience has grown, a selection of generalised algorithmic techniques has emerged, each well suited to a certain variety of problems but inappropriate for others. For example, we are familiar with concepts such as “divide and conquer”, “dynamic programming” and more recently “simulated annealing”. When investigating a new problem we may try to formulate a solution in one of these well known styles. Since we already know how to implement the essential computational structure of each technique, it will only be necessary to introduce problem specific details to produce a new, tailor-made program.

An analagous process can provide the foundation of our new system. We must identify a similar collection of algorithmic techniques, each having a computational structure which lends itself specifically to parallel implementation. Thus, if the essential structure can be parallelized, then any instantiations of the technique dealing with real problems will also share this property. This is already beginning to happen. In particular, the inherently parallel qualities of “divide and conquer” have been noted [22, 28, 14]. However, to produce the kind of idealised system envisaged above, we must go one step further than the development of a new set of parallel programming “folklore”, describing such good techniques. Instead, we must aim to embed them within the syntactic structure of the programming model in a way which forces the programmer to use one or other of them as the central structure of every program. Furthermore, we would ideally like the programmer’s visualisation of these structures to be non-parallel.

Our second source of inspiration suggests a means by which we can achieve this strict enforcement in a coherent, high-level way. This is the concept of “higher order” functions which is central to the expressive power of pure functional languages. We now present a brief introduction to “higher order functions”, for the unfamiliar reader.

Simple functions accept items of some data type as arguments and return items of some data type as results. For example, the integer square function defined

$$square\, x \;=\; x \,*\, x$$

takes an integer as its argument and returns an integer as its result. The type of this function can be denoted

$$square : int \rightarrow int$$

Some simple functions are able to operate upon items of more than one data type because the operations they perform do not inspect the details of the argument but only its “structure”. A good example is the function “*head*” which takes a list of items of some particular type as its argument and returns the first item on the list as its result. Clearly the type of the items themselves is of no consequence and such functions are said to be "polymorphic" [1]. Their type can be denoted using dummy symbols to represent arbitrary types. Thus, "head" has type

$$head : [a] \rightarrow a$$

where square brackets denote "list of" and "*a*" stands for any type.

Higher order functions are slightly more complicated. Rather than taking and returning simple data items, they accept other functions as arguments and return new functions as results. The function which is returned is a structural composition of the argument functions and possibly some others. Probably the simplest and most commonly used higher order function is "*map*". The single argument required by *map* is some function f with type

$$f : a \rightarrow b$$

Remember that a and b can be any, possibly identical, types. The result of applying *map* to such an f is another function "*map f*" of type

$$(map\ f) : [a] \rightarrow [b]$$

that is, a function which has a list of elements of type "*a*" as its argument and returns a list of elements of type "*b*". The effect of "*map f*" is to apply "*f*" to every element in the argument list, placing the result in the corresponding position in the result list. For example, "*map square*" would have type

$$(map\ square) : [int] \rightarrow [int]$$

and would return the list of squares of the integers in the argument list. The important point is that we can regard "*map*" as a function in its own right, with type

$$map : (a \rightarrow b) \rightarrow ([a] \rightarrow [b])$$

i.e. a function which deals with other functions rather than simple data types.

Higher order functions are precisely the kind of object which we seek to give a more solid basis to our notion of "generalised algorithmic techniques". They do not concern themselves with the lowest level details of particular problems. Instead, they capture the higher level computational structure of whole classes of algorithm. Solutions to particular problems can be instantiated by supplying the "customising" argument functions appropriate to the task in hand. Thus, we can equally well use *map* in conjunction with some other simple function to

[1] i.e. "having many forms"

produce a different result, using a method which has the same overall structure. For example, given a function "*uppercase*" which returns the uppercase equivalent of its lower case character argument

$$uppercase : char \rightarrow char$$

we can easily use "*map uppercase*" to transform a list of characters in the same way

$$(map\ uppercase) : [char] \rightarrow [char]$$

The high-level algorithm is identical to that of "*map square*" with only the lower level details differing. In other words, the use of "*map*" ties down the essential structure of our computation, while its argument function produces a problem specific program which solves the task in hand.

2.2 Algorithmic Skeletons

While higher order functions fit snugly into the syntactic framework of functional languages, we can also imagine a similar facility in the context of conventional imperative languages. Here, a higher order function could be represented as a program or procedure "template", specifying the overall structure of a computation, with gaps left for the definitions of problem specific procedures and declarations. The fit would be somewhat better in languages which allow procedures to be passed as parameters. Thus, the system which we are about to propose could be presented to the programmer in the context of any "base" language, whether declarative or imperative. Our subsequent use of the term "higher order function" should be interpreted with this freedom in mind.

The programming model which we intend to investigate can now be described. In this model, the programmer is presented with a selection of specialised higher order functions (or similar, depending upon the base language) from which one must be selected as the outermost function in the program. The programmer may then use the full power of the language to describe the functions to which the selected higher order function will be applied to produce the problem specific final program. The restriction on program structure imposed at the highest level is the means by which we curb the "universality" problem. Only those programs whose outermost structure matches one of the acceptable higher order functions are legal. Having achieved this restriction, we will not be concerned with the lower level details of program, since these will not affect the essential structure of the algorithm or its implementation.. Thus, the full power of the "host" language can be made available at these levels. We will refer to each suitable higher order function as an "algorithmic skeleton", since it describes the computational skeleton of an algorithm without overspecifying the details.

It would be quite easy to provide such a facility in a conventional environment, with no thought to parallelism. We would define "skeletons" for all the familiar algorithmic paradigms and support each with an optimised machine code implementation into which the problem specific user code would be plugged automatically. The key to achieving efficient parallel performance for such a model is to ensure that each of the permissible higher order functions describes a computational structure for which we can find an efficient parallel implementation, irrespective of the problem specific details. In other words, it should correspond to a good parallel "algorithmic technique". Of course, the actual specification of each skeleton as seen by the programmer need not (and ideally should not) be explicitly parallel. It is sufficient that it be equivalent to some other explicitly parallel "skeleton" which the system knows how to implement.

If we are to avoid falling into the trap of trying to implement a universal language automatically, it seems that we must restrict ourselves to exploiting only the parallelism inherent in the basic structure of each skeleton. The lower level problem specific functions may or may not be parallelizable, but we should ignore them, since to attempt deeper analysis leads us into the the original trap of "universality". The implementation task is to parallelize the distribution and manipulation of data implied by the highest level structure, leaving other functions to be executed entirely sequentially on individual processors as required, just as user code would slot directly into the structure of an entirely sequential implementation. This approach has two subsidiary benefits. Firstly, the choice of language used to describe the problem specific details is only restricted by the availability of compilers for the sequential processors which comprise the parallel machine. Secondly, we can always take advantage of the best such compiler, since the generated code is not required to interact with the parallel super-structure.

In summary, the programmer sees each skeleton as a higher order function (or program "template" if the base language is imperative) which is one of a collection of algorithmic tools from which a problem specific program must be fashioned. Meanwhile, to the system implementor, each skeleton is a generic computational pattern for which an equivalent, efficient parallel "harnesses" must be defined. Just as conventional languages require different compilers for different machines, so each skeleton will require a different implementation for each parallel machine. As with conventional compilers, these differences will be invisible to the programmer, who always sees the same machine independent programming framework.

In order to contrast our approach to those discussed earlier, it is instructive to consider the ways in which the issues of problem decomposition and distribution are handled. The key observation is that the parallel decomposition is already implicit in the structure of each skeleton. Thus, in selecting a particular skeleton (which may either be explicitly declarative or imperative and sequential), the programmer is also unwittingly selecting a parallel problem decomposition. Similarly, the task of distribution has been reduced to the problem of implementing each of a selection of pre-defined decomposition patterns, one per skeleton. Since these must only be implemented once (for each hardware topology), they can be carefully hand-crafted to extract maximum performance. All complete programs written in terms of the same skeleton will use the same pre-defined implementation structure with its in-built distribution strategy. Thus, in selecting a skeleton, the programmer is also selecting an associated distribution plan.

We may throw further light on the proposal by imagining an interactive session with a "programming environment" which could be constructed around such a model. Upon activating the system, the programmer might be 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 would be selected. Similarly, the programmer would select a base language in which to specify the procedures and data structures fleshing out the skeleton. The system would respond by displaying the generic program which describes the operation of the selected skeleton in the chosen base language.

Finally the system prompts the user to provide descriptions 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 user code fragments are loaded across the parallel hardware together with the pre-defined 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, calling up user routines as appropriate, 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.

A useful comparison can be made with the "routine library" facility, often found as part of a conventional programming model. Here, 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 program 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) frameworks. The programmer's task is to select one and fill in the details. Our long term 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 language.

2.3 Parallel Hardware

If our proposed system is to be shown to be genuinely useful, it will not be sufficient to show that each skeleton encapsulates a good parallel problem decomposition. It will also be necessary to show that an efficient distribution of this parallelism can be implemented on a realistic parallel machine. Subsequent chapters will sketch and analyse such implementations, which will all be described in terms of one machine model. This section introduces our choice of this underlying hardware. It must be emphasised that this selection and the corresponding implementation represents only one possible instantiation of our "skeletal machine". There is nothing inherent in the high level specifications which ties us to any particular model of parallel hardware. Many other possible implementations can be envisaged. Our purpose here is simply to show that one unarguably realistic implementation could be constructed.

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 designer of programming systems. 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 systems be designed in such a way that substantial hardware 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 text 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. However, to provide coverage of sufficient depth for the our purposes, 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 [25] is constructed from a collection of independent 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 the New York University Ultracomputer [12] 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. In our selected hardware we take the former approach of associating each processor with a unique memory module. We will attempt to exploit this locality in our implementations.

The other important area concerns the network structure and its degree of integration with the 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. In terms of integration we must decide whether processors and memory should be distributed throughout the network and active within it, or if they 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. Once more, for our purposes, a decision is made in favour of its presence. 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 our 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 of this decision for the choice of a particular network is that the level of connectivity between physical elements must be sparse. Bearing in mind that we expect the number of processors and memories in use to be large, it is not reasonable to expect a single component (e.g. a bus) to connect directly to many other components and still provide reasonable performance. A wide variety of networks have been proposed as a basis for general purpose parallel computers, with consideration given to issues such as degree, diameter, optimal layout, wire length, regularity and fault-tolerance. The requirement for sparsity suggests that we should only consider networks whose degree grows as a small function of the number of elements, or even only those in which the degree is constant. Furthermore, the poor results for two dimensional layout, and consequently edge length, of most "logarithmic diameter" networks (e.g. as reported in [6]) make networks based on hypercubes, shuffles, etc. less attractive if we are looking towards direct hard-wired implementation as a long term goal. 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 a 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[2] 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 this model is unarguably realistic and practical. Other networks mentioned could equally well be used.

Figure 2.1 shows a small instance (with 16 processors) of the selected hardware[3]. 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 transputer [19], it is assumed that the processor instruction set includes "send"

[2]And for that matter, fault tolerance, although this is not considered here.

[3]For convenience, subsequent illustrations shrink the connecting links to have zero length, and represent the hardware as a simple grid.

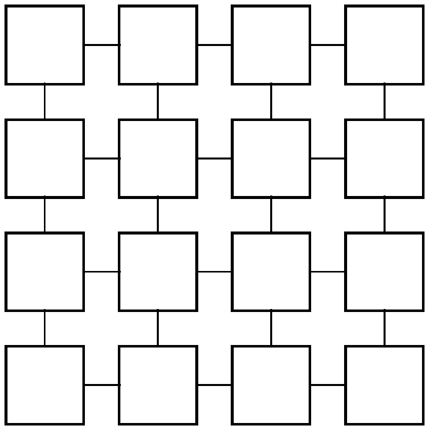

Figure 2.1: A sixteen processor grid

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 follow the occam model [18]. 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 broadly 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[4].

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.

[4]Although care must be taken to avoid deadlock, as in any parallel machine.

2.4 Implementation Structures and Performance Measures

2.4.1 Building Skeleton Implementations

It has been emphasised in the preceding discussion that the skeletons comprising the overall framework are independent. Consequently, their implementations can be considered separately, as indeed they are for the four skeletons presented in chapters 3 to 6. This independence illustrates another useful property of this 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.

The implementation of each skeleton must ensure that the grid performs 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, an appropriately parameterised 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, at some point in a computation, 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[5].

[5]The details of what lies beyond the grid are beyond our scope. 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 crucial problems addressed here concern the control and manipulation of data required to

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 data between processors. As will become clear in subsequent chapters, such communications may take two forms, being either 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 2.3 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, in any such case, 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.

2.4.2 On Measuring Performance

2.4.2.1 A Note on Notation

The $0, \Omega, \Theta$ notation introduced by Knuth [21] is used throughout the text to indicate the asymptotic magnitude of various quantities. Thus, the statement

$$f(n) = O(g(n))$$

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,

$$f(n) = \Omega(g(n))$$

produce solutions efficiently.

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".

2.4.2.2 Performance Issues

In terms of performance, our interest is focussed on the efficiency with which a large grid of processors can implement each skeleton 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 at the nodes of the square grid should be identical to the uniprocessor machine against which their performance will be considered. This assumption has two implications which should be borne in mind when interpreting the performance analyses presented subsequently.

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 where m is 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.

This 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 not pose any serious difficulty.

2.5 Related Work

As was illustrated in chapter 1, the bulk of the work in the field addresses the problems of implementing universal languages, whether declarative or explicitly parallel. However, we can also identify a number of approaches which are more in the direction proposed here.

The most closely related work has focussed on the "divide and conquer" technique and its parallel implementation as a tree of processes. In particular, [22] characterizes a general form of "divide and conquer" as a higher order function and reports on a parallel implementation which dynamically maps the problem dependent process tree across a network of transputers. We will compare this work with that of our own skeletons in more detail in chapter 3.

In similar vein, [24] describes a "generalised combinatorial search algorithm" as a framework from which a variety of more specialized combinatorial algorithms can be derived. Once again, the intention is that particular instances can then be related to appropriate parallel implementations. Elsewhere, [27] reports on practical experiments involving the parallel implementation of "problem heap" algorithms, which are essentially another very general formulation of the divide and conquer approach.

In the occam world, several general purpose "harnesses" have been implemented. Typically, these are still at a fairly low level and require the programmer to deal with parallel problem decomposition explicitly, but provide support for communication and some simple distribution. Similarly, [5] reports on the implementation of a variety of "monitors" which provide machine independent support for a variety of communication and synchronisation primitives. Again, the task of explicit problem decomposition remains with the programmer.

Chapter 3

The Fixed Degree Divide & Conquer Skeleton

3.1 Introduction

The first skeleton to be considered, is a variation of the well known "divide and conquer" technique (e.g. see [1]), which itself needs almost no introduction. In its most general form, "divide and conquer" is applicable 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 problem instance is "indivisible" in some sense, in which case a direct solution is obtained by some other simpler method.

Divide and conquer algorithms clearly offer good potential for parallel evaluation. It is not difficult to see that recursively defined sub-problems may be evaluated concurrently if sufficient processors are available. The whole execution of a divide and conquer algorithm amounts to the evaluation of a dynamically evolving tree of processes, one for each sub-problem generated. The challenge is to ensure that the mapping of this virtual tree to the real machine is performed as efficiently as possible. The task of parallelising the divide and conquer paradigm in its loosest form has been considered elsewhere [28, 14, 22]. Before introducing our own more restricted version and its implementation, we consider one of the more recent of these approaches in more detail, since its target machine bears some resemblance to our own.

3.2 The ZAPP Approach

ZAPP[22] (Zero Assignment Parallel Processor) is a virtual tree machine which dynamically maps a process tree (e.g. as produced by a divide and conquer algorithm) onto any fixed network of processor-memory pairs (i.e. with no shared memory, as in the target hardware chosen here). The fundamental principle involved in producing the mapping of processes to processors is that of "stealing", by which a processor may obtain a new problem (process) to work on from one of its physical neighbours. Distribution of problems is therefore "demand driven" - a processor will attempt to solve a problem and all of its sub-problems, unless these are stolen for evaluation elsewhere. The pattern of stealing is controlled by the "single steal rule" which says that a "stolen" problem may not be subsequently stolen again. Thus, such a problem is always solved on a processor physically adjacent to that of its parent problem, thereby avoiding the potentially substantial communication overheads of a less restrictive method. On the other hand, the single steal rule means that diffusion of the whole problem through the machine is a slower process than might otherwise be achieved. Preliminary experiments suggest that the technique is most successful when the number of sub-problems available is substantially larger than the number of processors[22].

The ZAPP version of divide and conquer presents the technique, in its loosest form, as a higher order function "D_C", defined to operate as

$$\begin{aligned} D\_C\ indivisible\ split\ join\ f &= F \\ \text{where } F\ P &= f\ P, \ \text{ if } indivisible\ P \\ &= join\,(map\ F\ \ (split\ P))\,, \ \text{otherwise} \end{aligned}$$

Thus, if problem instances are of type "*prob*" and solutions are of type "*sol*", then the programmer produces a problem specific program from the higher order function by providing definitions of

$$\begin{array}{ll} indivisible & : prob \rightarrow boolean \\ f & : prob \rightarrow sol \\ split & : prob \rightarrow [prob] \\ join & : [sol] \rightarrow sol \end{array}$$

where "*indivisible*" is a function which inspects a problem instance and decides whether it can be solved recursively, "*f*" is the straightforward function for directly solving indivisible (or "base case") instances, "*split*" is the function which decomposes a "difficult" problem into several sub-problems and "*join*" is the function which describes how to combine the solved sub-instances to solve the instance from which they originated. The higher order function D_C has type

$$\begin{aligned} D_C : \ \ & (prob \rightarrow bool) \rightarrow (prob \rightarrow [prob]) \rightarrow ([sol] \rightarrow sol) \\ & \rightarrow (prob \rightarrow sol) \rightarrow (prob \rightarrow sol) \end{aligned}$$

and so the type of a full program with all of its "customising" functions specified as required is $prob \rightarrow sol$ as we would wish.

It is important to note that this version of divide and conquer makes no assumptions about the shape of the process tree which will be produced by specific problems or problem instances. In particular, the number of sub-problems produced upon "splitting" may vary from node to node as may the depths of different sub-trees. Since the structure of the process tree only emerges dynamically, it is impossible to pre-assign a distribution pattern of processes to processors. The ZAPP "stealing" approach aims to allow a problem specific (and physical topology specific) distribution to evolve naturally, with work generated in "heavy" sections of the tree diffusing through the network to processors which have finished dealing with "light" branches. A variety of refinements of this basic strategy are discussed in [22].

3.3 Fixed Degree Divide & Conquer

The ZAPP version of divide and conquer fits our description of an "algorithmic skeleton". It has a non-parallel specification as a problem independent higher order function and an invisible parallel implementation. However, while the results reported are encouraging, the unpredictable nature of the process tree and the consequent generality of the implementation technique are somewhat looser than the rigidly pre-defined style of distribution that we have in mind for our own skeletons. Consequently, we now propose a more restrictive formulation of divide and conquer which will allow us to take a firmer grip of distribution at the expense of a degree of flexibility at the programmer's level. Such an approach is clearly in sympathy with the original motivations discussed in chapter 2.

The restriction we add requires that for any particular divide and conquer program, the degree of all the non-leaf nodes in the process tree is a constant which is known before execution begins. In other words, we require that any problem which is not solved directly will produce some constant k sub-problems upon splitting. To use the skeleton, which we will call "fixed degree divide and conquer" (FDDC), the programmer must specify the value of k and specify the functions "*indivisible*", "f", "$join_k$" and "$split_k$", where the splitting and joining functions respect the value of "k" in their operation. With this added restriction, the structure of the skeleton as it appears to the programmer is identical to that presented for the ZAPP machine.

The simplest and commonest FDDC 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 3.4 considers such binary FDDC 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 3.4 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 some concrete examples.

3.4 Implementing the Skeleton

3.4.1 An Idealised Implementation

The FDDC skeleton, as presented in section 3.3 has an obvious parallel implementation in which the k-ary tree of processes is mapped directly onto a k-ary tree of processors.

The root processor is presented with the whole problem instance which it subdivides. 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 "f" 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.

3.4.2 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. 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 3.4.1 results in an implementation of the FDDC skeleton for which the number of inactive processors becomes negligibly small as

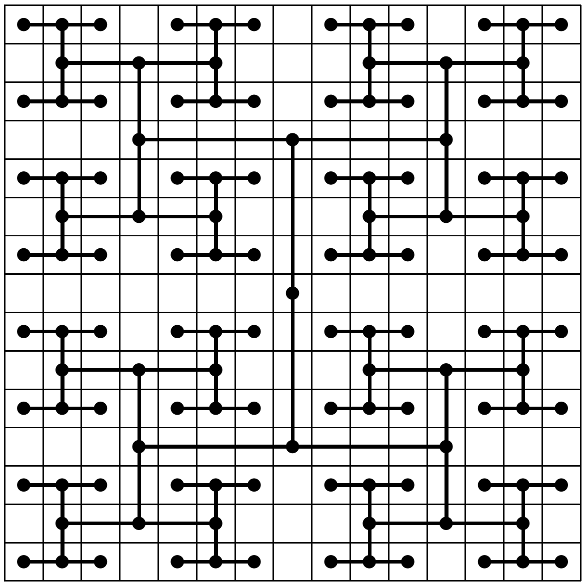

Figure 3.1: The H-tree layout.

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

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

$$\sum_{d'=0}^{\frac{1}{2}\log_2\left(\frac{v+1}{2}\right)-1} 2^{\frac{1}{2}\left(\log_2\left(\frac{v+1}{2}\right)-2d'-2\right)} = \left(\frac{v+1}{2}\right)^{\frac{1}{2}} - 1.$$

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

$$4\left(\frac{v+1}{2}\right) - 4\left(\frac{v+1}{2}\right)^{\frac{1}{2}} + 1 = \Theta\left(v\right).$$

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. •

3.4.3 A More General Technique − $k = 4$

As noted in section 3.1 natural algorithms for many problems might be better described in terms of a $k = 4$ (or more) FDDC 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 ≥ 5, no direct counterpart of the H-tree exists. However, the layout in figure 3.2 shows that with controlled, regular edge sharing, the $k = 4$ tree can be embedded into the grid in linear area. Furthermore, it will become clear in section 3.5 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.

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

All processors are marked with a · 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.

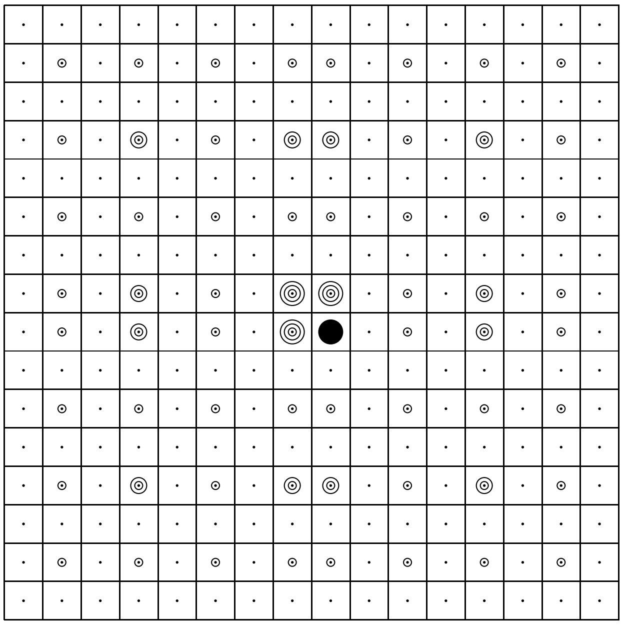

Figure 3.2: The 4-ary tree layout.

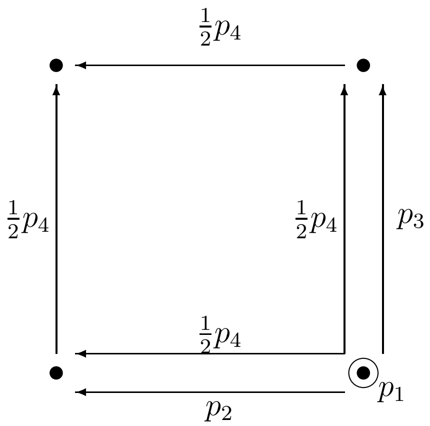

Figure 3.3: Problem distribution in the 4-ary tree.

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 $p1, p2, p3, p4$ with $p1$ 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 $p4$ first allows the second halves of their journeys to be overlapped in time with the distribution of problems $p2$ and $p3$.

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, as a simple function of its absolute location in the grid, whether it should be active in the computation, simply forwarding information or inactive.

3.4.4 Completely General k

The essence of the technique used to handle the case $k = 4$ was to recognize that we could sub-divide the original grid into four identical square sub-regions within which we could recursively and independently handle the four sub-problems generated at the root. It is not difficult to see that the same technique may be conveniently extended to deal with any k which is a perfect square. Thus, for $k = 9$, we will recursively sub-divide into nine independent sub-squares, and so on. For a tree with 9^n leaves we will use a $9^{\frac{n}{2}} \times 9^{\frac{n}{2}}$ square grid in a neat, regular way.

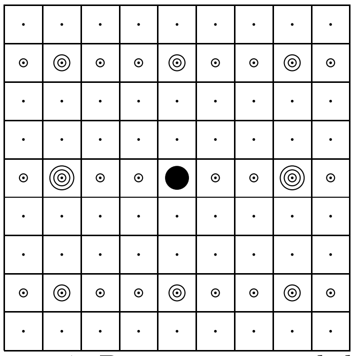

Figure 3.4: Partitioning with k=3

What of other values of k ? The simplest version of our technique does not apply directly. For example, we cannot divide a square grid regularly into five equally sized sub-squares. However, a simple arithmetic observation comes to the rescue. For any k, we are attempting to map the k^n leaves of the tree onto k^n processors. However, for any k, n note that

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

where, of course, k^2 is a perfect square. Thus, if n is even, making $\frac{n}{2}$ an integer, we can adopt a very similar approach to the case where $k' = k^2$ with half as many levels in the tree. To allow for the extra levels we need for the real k, we simply split each level in the k^2 case in two, partitioning the grid first in (say) the "vertical" direction, then horizontally. For example, with $k = 3$ and $n = 4$, we partition first into three vertical slices and then split each of these horizontally into three 3×3 single processor sub-grids, for a total of 81 processors, which is what we want. The result is illustrated in figure 3.4 with the number of circles indicating activity at each level of the tree, as before. Figure 3.5 illustrates the paths followed by two typical sub-problems. The problem destined for the top left-hand processor is passed on during every distribution phase. The one which arrives in the lower half of the grid is the sub-problem allocated to the root for the first two levels, before being dispatched.

Finally, for situations in which n is odd, we can initially attack the problem as though n were one larger and leave out the final (meaningless) level of decomposition. Thus, we actually employ k^{n+1} processors, of which only k^n do useful work. However, as with the analogous case for the H-tree, the factor of k is lost in subsequent asymptotic analyses.

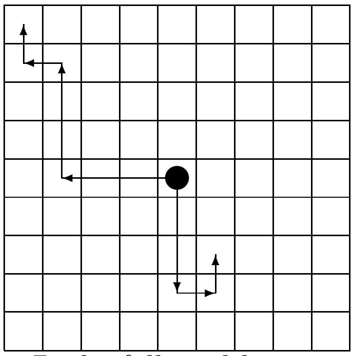

Figure 3.5: Paths followed by two problems

3.5 Analysis of Implementations

In this section we consider the performance which can be expected of the proposed implementations. The abstract specification 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 3.7 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), and that the resulting tree is well balanced. 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 [1].

In this context, the notion of size may have two connotations. It could describe either the quantity of data required to represent a sub-instance, or the amount of further divisibility inherent in that sub-instance. 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 n 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 will eventually sub-divide 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 3.6

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 processor) is obtainable. Finally, the existence of similar results for the generalised implementation technique of section 3.4.4 is noted.

3.5.1 The Full Binary Tree

We now consider the performance of the H-tree implementation of the binary FDDC 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. an instance 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

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

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

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

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 $f\left(1\right)$,

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

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

The execution time of this parallel implementation of the skeleton, denoted $T_{P2}\left(n\right)$ 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

$$\begin{aligned} T_{P2}\left(n\right) &= 4\left(\sqrt{n} - 1\right) + f\left(1\right) - 2\log_2 n \\ &\quad + \sum_{i=1}^{log_2 n} \left[s\left(2^i\right) + j\left(2^i\right) + p\left(2^{i-1}\right) + r\left(2^{i-1}\right)\right] \end{aligned}$$

for cases in which the depth $\log_2 n$ is even. When the depth is odd, the first term is replaced by $3\sqrt{2n} - 4$.

3.5.2 Optimal Efficiency

We now consider conditions under which the $\Theta\left(n\right)$ 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 *If any of $s\left(n\right)$, $j\left(n\right)$, $p\left(n\right)$ and $r\left(n\right)$ are $\Omega\left(1\right)$, then*

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

Lemma 2

$$h\left(n\right) = O\left(n^a\right), a > 0, \Rightarrow \sum_{i=1}^{log_2 n} h\left(2^i\right) = O\left(n^a\right).$$

Lemma 3

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

A straightforward sequential evaluation would (ignoring constant overheads for handling recursion) perform all the "real" operations of each vertex in the processor tree. This excludes those associated purely with data transfer, which are redundant. Thus, the time taken by such an implementation would be

$$T_{S2}\left(n\right) = nf\left(1\right) + n\sum_{i=1}^{log_2 n} \frac{1}{2^i}\left[s\left(2^i\right) + j\left(2^i\right)\right].$$

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,

$$1 \cdot T_{S2}\left(n\right) = \Theta\left(nT_{P2}\left(n\right)\right).$$

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 $T_{S2}\left(n\right) = O\left(nT_{P2}\left(n\right)\right)$ (it executes the same operations without communication overheads) it is only necessary to investigate

$$T_{S2}\left(n\right) = \Omega\left(nT_{P2}\left(n\right)\right)$$

i.e.

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

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

The $nf\left(1\right)$ term on the right can be removed as, by lemma 1, can the $n\sum\frac{1}{2^i}\left(\ldots\right)$ term (provided that we are not dealing with a problem in which all of the tasks represented by s, j, p and r get easier as the grain increases! Do any such problems exist?).

This leaves

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

with one of s, j, p, r at least constant, 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.

In other words, 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

$$T_{P4}(n) = f(1) + \sum_{i=1}^{\log_4 n} \left[s\left(4^i\right) + j\left(4^i\right) + \frac{3}{2}\left(p\left(4^{i-1}\right) + r\left(4^{i-1}\right)\right) + 4\left(d(i) - 1\right) \right]$$

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 $1 \le i \le \log_4 n - 1$ and $d(\log_4 n) = 1$ the d's are simplified to obtain

$$T_{P4}(n) = f(1) + 2\sqrt{n} - 4\log_4 n + \sum_{i=1}^{\log_4 n} \left[s\left(4^i\right) + j\left(4^i\right) + \frac{3}{2}\left(p\left(4^{i-1}\right) + r\left(4^{i-1}\right)\right) \right]$$

By analogy with the binary tree it can be seen that

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

It is a simple exercise to show that similar results to lemmas 1-3 and their consequences hold.

Turning finally to the implementation for completely general k, 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. Since the grid is square and no path from processor to leaf is longer than that from root to top-leftmost processor, which does not double back on itself, the total contribution of edge delays on any direct root to leaf path is

$$\sum_{i=1}^{\log_k n} (d(i) - 1) = O(\sqrt{n}).$$

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 within distinct subtrees. It is now clear that even the simplest procedure, in which a parent distributes 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 our asymptotic results extend to any k.

3.6 Partial Trees

The drawback of implementation on a complete tree 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 most 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. Equally, it is unreasonable to assume that any fixed size machine will be large enough to handle, in this 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 of the process tree at higher levels of the processor tree. 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 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 of course, the number of processors used 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

$$\begin{aligned} T^c_{P2}(n) &= \sum_{i=c+1}^{\log_2 n} \left[s\left(2^i\right) + j\left(2^i\right) + p\left(2^{i-1}\right) + r\left(2^{i-1}\right) \right] \\ &\quad + 2 \sum_{i=1}^{\log_2\left(\frac{n}{2^c}\right)} \left[d(i) - 1 \right] + 2^c \left(f(1) + \sum_{i=1}^{c} \frac{1}{2^i} \left[s\left(2^i\right) + j\left(2^i\right) \right] \right) \end{aligned}$$

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

$$T_{S2}(n) = \Omega\left(\frac{n}{2^c} T^c_{P2}(n) \right)$$

need be considered.

Removal of redundant terms leaves

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

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

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. Take $c = C\log_2 n$, with $0 < C < 1$, to indicate 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

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

and thus when ,

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

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 $O(n^a)$ for some $a > 0$. If $a \leq C$ then analysis similar to the full tree case shows that

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

gives optimal efficiency. With $a > C$

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

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

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

If $a > C$ then it may be simply shown (using lemma 3) that the second term is never sufficient to outweigh n^{a-C} and $f(1) = \Omega\left(n^{a-C}\right)$ 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. In other words, activity at the idealised leaves ensures optimal efficiency (in our asymptotic sense) whenever the exponent of the upper bound on s, j, p and r is no larger than the fraction of levels which are executed by simulation, when this itself is at least $\frac{1}{3}$. The important point is that the work required of the idealised leaves to ensure optimal efficiency is no longer dependent upon n, as it was for the full tree implementation.

Two of the examples presented in the next section have characteristics which allow them to exploit this result.

3.7 Examples

3.7.1 The Discrete Fourier Transform

An algorithm to compute the discrete fourier transform (DFT) of n points, as presented in [32] can be described very neatly with the simple $k = 2$ FDDC skeleton. 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

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

$f(1) = 1$, the leaf operation being a single multiplication,

Thus,

$$\begin{aligned} T(n) &= 4(\sqrt{n} - 1) + 1 - 2\log_2 n + \sum_{i=1}^{log_2 n} \left[0 + \frac{3}{2}\left(2^i\right) + 2^{i-1} + 2^{i-1}\right] \\ &= 5n + 4\sqrt{n} - 2\log_2 n - 8. \end{aligned}$$

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 FDDC 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 adaptation of the well known algorithm in [30]). However, [6] 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 FDDC discrete fourier transform 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 shared memory 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 [34]). The whole algorithm can therefore be executed in time $\Theta(\sqrt{n} \log n)$, using $\Theta\left(n^{\frac{3}{2}} \log n\right)$ resources, which is superior to both other methods. However, the FDDC solution is certainly the most easily specified. Furthermore, it is very important to note the FDDC skeleton assumes that the input is initially located at a single processor, thereby 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". Since execution time is dominated by activity at the root, the observations on implementation by partial tree cannot be usefully employed here.

3.7.2 Approximate Integration

The approximate integration of some continuous function y over an interval $[a, b]$ is an obvious candidate for evaluation using the FDD C skeleton, given the observation that

$$\int_a^b y(x)\ dx = \int_a^{\frac{b-a}{2}} y(x)\ dx + \int_{\frac{b-a}{2}}^b y(x)\ dx.$$

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 δ at or below which an approximation to the integral will suffice (thus requiring no further sub-division) i.e.

$$\int_{c}^{c+\delta} y(x)\ dx = approx(y, c, \delta).$$

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

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

- $s(n) = 2$, the cost of calculating $\frac{b-a}{2}$,
- $j(n) = 1$, 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 3.5 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. More realistically, suppose that $f(1)$ is a constant. Plugging the values of s, j, p and r into the formula for parallel run time produces

$$T_{P2} = O\left(n^{\frac{1}{2}}\right).$$

Since leaf evaluations are trivial, run time is dominated by the distribution costs. This fully parallel implementation uses $O(n)$ processors. In contrast, using the formula for sequential time, we obtain a one processor implementation with run time

$$T_{S2} = O(n).$$

and the inefficiency of the fully parallel version is apparent.

However, since all of s, j, p and r are $O\left(n^{\frac{1}{3}}\right)$, our analysis of evaluations on partial trees indicates that optimal efficiency can be achieved provided that

C, the the fraction of lowest levels of the virtual process tree implemented by simulation, is at least $\frac{1}{3}$. Thus, beginning with a single processor implementation ($C = 1$) with $O(n)$ time and "$O(1)$" processors, we can progress through a series of implementations in which more levels are implemented with physical parallelism (decreasing C) and still maintain optimal efficiency, with $O\left(n^{C}\right)$ run time on $O\left(n^{1-C}\right)$ processors. This progression ends when $C = \frac{1}{3}$, when the run time is $O\left(n^{\frac{1}{3}}\right)$ on $O\left(n^{\frac{2}{3}}\right)$ processors.

Curiously, decreasing C further results in increases in both run time (which is now $O\left(n^{\frac{1-C}{2}}\right)$) and processors used (still $O\left(n^{1-C}\right)$! The increasing run time in this final phase of expansion is explained by the fact that the overall run time is now dominated by the cost of distributing small parcels of work to many distant processors.

3.7.3 Matrix Multiplication

The third example, matrix multiplication, uses the generalised FDDC skeleton. The algorithm presented here is adapted from [14].

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

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \cdot \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix} = \begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix}$$

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 $\frac{m}{4}$ element matrix multiplications and four $\frac{m}{4}$ element matrix additions. This is just an FDDC 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 3.5, 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 the 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 $n^{\log_8 4}$, which is $n^{\frac{2}{3}}$, elements. In terms of the measures of section 3.5 this gives, $p(n) = 2n^{\frac{2}{3}}$, the two input matrices, $r(n) = n^{\frac{2}{3}}$, $s(n) = 0$, $j(n) = n^{\frac{2}{3}}$ and $f(1) = 1$. Note that since $s(n)$, $j(n)$, $p(n)$ and $r(n)$ are all $O\left(n^{\frac{2}{3}}\right)$, the "optimal efficiency" result for partial trees will be applicable.

First we consider the execution time for a full tree implementation. Using the simplest problem distribution method, by which a processor at one level distributes sub-instances sequentially, the time for m element matrices is

$$\begin{aligned} &= 2\left[8\left(\frac{m}{4}\right)+8\left(\frac{m}{4^2}\right)+...+8+m^{\frac{3}{4}}\right]+1 \\ &\quad +4+16+...+\frac{m}{16}+\frac{m}{4}+m \\ &= O(m). \end{aligned}$$

The terms preceded by 8's represent the transfer of information, the $m^{\frac{3}{4}}$ is the delay across the $m^{\frac{3}{2}}$ processor grid, the 1 is the base case calculation and the other terms represent the additions involved in combining sub-results. Reformulating in terms of the grain of problem instances produces the result that the parallel run time for the $k=8$ skeleton is

$$T_{P8} = O\left(n^{\frac{2}{3}}\right)$$

For a sequential implementation, the time taken for instances involving size m matrices, $S(m)$, is described by the recurrence relation

$$\begin{aligned} S(m) &= 8S\left(\tfrac{m}{4}\right)+m, \\ S(1) &= 1. \end{aligned}$$

which has solution $S(m) = O\left(m^{\frac{3}{2}}\right)$, the same as the standard sequential matrix multiplication algorithm. Thus, since these matrices are actually of grain $n = m^{\frac{2}{3}}$, the sequential run time for matrices of grain n is

$$T_{S8}(n) = O(n).$$

Thus, for instances of grain n, the full parallel implementation achieves an $O\left(n^{\frac{1}{3}}\right)$ speed up at the expense of using $O(n)$ processors. This is not optimally efficient. However, the fact that $s(n)$, $j(n)$, $p(n)$ and $r(n)$ are all $O\left(n^{\frac{2}{3}}\right)$, indicates that a partial tree implementation can achieve optimal efficiency provided that $C \geq max\left\{\frac{1}{3},\frac{2}{3}\right\}$. In other words, if we choose to simulate at least the lower $\frac{2}{3}$ of the virtual process tree at the leaves of the grid-embedded physical tree, then the execution will be optimally efficient.

Thus, starting with a one processor implementation (simulating the the whole tree in $O(n)$ time with "$O(1)$" processors), we can, up to a point, add real processors at successively deeper levels to give an asymptotically optimal speed up. More precisely, if we still simulate the lower $C\log_8 n$ levels (thereby using $O\left(\frac{n}{8^{C\log_8 n}}\right) = O\left(n^{(1-C)}\right)$ processors) we obtain a run time of $O\left(n^C\right)$. This approach reaches its limit when $C=\frac{2}{3}$, at which point the run time is $O\left(n^{\frac{2}{3}}\right)$ with $O\left(n^{\frac{1}{3}}\right)$ processors. Implementing levels below this with real parallelism simply uses more processors without affecting the run time asymptotically.

Chapter 4

The Iterative Combination Skeleton

4.1 Abstract Specification

4.1.1 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, one style of 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", 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 of this approach, consider Sollin's algorithm [33] 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 single vertex 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

```
WHILE |S|<>1 AND NOT failure to find any combinations DO BEGIN
  FOR each s in S
    find s' in S  such that s' is the "best" partner for s
    by considering all possible partners in turn;
  combine ''best partners'' to reduce |S|
END
```

Figure 4.1: Imperative Specification

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" (IC) skeleton models this style of solution. Its abstract specification and possible parallel implementation on the grid machine are presented in this chapter.

4.1.2 Specification

It is a question of personal taste as to whether the IC skeleton is most naturally specified in an imperative style, as a sequential program "template", or in a declarative style as a higher order function. To give as much insight into its meaning as possible, both approaches are presented here. We begin with the looser, more operational imperative specification, before presenting the more formal declarative version.

To the imperative programmer, the IC skeleton may be seen to implement the "pseudo-code" presented in figure 4.1, where S represents a set of homogeneous objects describing a problem instance.

In order to tailor the skeleton to a particular problem, the user must provide a type specification of an "object", 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.

More formally, if the objects involved are of type "*obj*", then the programmer must provide functions

$$\begin{array}{lll} combine & : & obj \rightarrow obj \rightarrow obj \\ value & : & obj \rightarrow obj \rightarrow val \\ accept & : & val \rightarrow val \rightarrow bool \end{array}$$

where *combine* describes how two objects are merged, "*value*" measures the utility (of type "*val*") to the overall solution of such a merger and *accept* discriminates between two such values, to select the most suitable (i.e. should the first value be selected in preference to the second). Given these functions, the IC higher order function is specified so that

$$\begin{aligned}
IC\ combine\ value\ accept\ &= F \\
\text{where } F\ S\ &= \ F\ (merge\ (partners\ S)\ S),\ \text{if } continue\ S \\
&= S,\ \text{otherwise} \\
merge\ &= ....combine.... \\
partners\ &= ....combine....value...accept \\
continue\ &= ....accept....value
\end{aligned}$$

The functions *merge*, *partners* and *continue* are part of the imaginary implementation, with *continue* using the programmer's functions *accept* and *value* to decide whether recursion should continue, *partners* using *combine*, *value* and *accept* to determine the set of "best partner" pairs for the given set of objects, and "merge" using "*combine*" to update the set of objects on the basis of these pairings. Using the notation $\{a\}$ to denote a "set of a" (where a fully executable specification might have to use lists)[1], the types of the internal functions are

$$\begin{aligned}
merge\ &:\ \{(obj, obj)\} \to \{obj\} \to \{obj\} \\
partners\ &:\ \{obj\} \to \{(obj, obj)\} \\
continue\ &:\ \{obj\} \to bool
\end{aligned}$$

It is not difficult to conceive of examples in which the best partner relationship is not necessarily symmetric i.e. in which, at some iteration, an object s_1 may be the best partner for s_2 but have s_3 as its own best partner. Indeed, this is the case with the minimum spanning tree algorithm. In such examples, all three objects involved in the "group" must be combined and replaced by a single object. Furthermore, it is essential that the process of combining such a group of objects follows a "valid" order in performing the combinations. For example, if tree t_1 has best partner t_2 via edge e, while t_2 has best partner t_3 via edge e', then any implementation must avoid combining t_1 directly to t_3 by some other spurious edge e'', before including t_2. In other words, objects must only be combined if they are directly best partners or have already subsumed other objects which imply a best partner link between them. The parallel implementation introduced subsequently respects this property.

[1] An executable sequential version of the skeleton also requires some further details, such as the labelling of objects, which do not serve to aid clarity and have consequently been omitted from this presentation. However, the author has successfully constructed such fully operational version in (sequential) Miranda[35]. Miranda is a trademark of Research Software Ltd.

4.2 Parallel Implementation Issues

The parallel implementation discussed in this chapter attempts to exploit the most obvious sources of parallelism in the imperative specification, namely the "for each s in S" loop and the "combine" phase. A sequential implementation would have to deal with objects in S one at a time in both cases. 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 we outline an actual grid implementation which addresses the problems raised.

4.2.1 Distribution of Valid Information

The composition 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.

4.2.2 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 case 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.

4.2.3 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 many-object "group" combinations may occur complicates the situation. The implementation must ensure that only one new object is created, replacing all its components and that any essential sequencing is obeyed within the construction of each new object. 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.

4.2.4 Balancing the Load

Each iteration alters the number of remaining objects and their respective sizes. 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 dynamic adjustments to the balance of work across the machine in order to preserve efficiency.

4.3 Implementation on an Idealised Grid

This section presents techniques which could be used to implement each iteration of the "test and select" loop and the combine" phase. 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 4.4 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 functions 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, and can therefore be implemented once and for all as part of the skeleton's underlying structure.

It will be assumed that 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, and are entirely independent of any identification introduced by the programmer.

4.3.1 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. We will present such a technique and then show how the ring (and therefore the implementation) can easily be embedded in the grid machine.

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, 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. The task of moving from ring to grid is now considered. Theorem 2 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 follows a

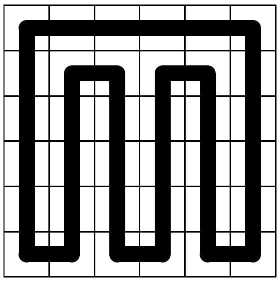

Figure 4.2: Hamiltonian circuit of an even-sided grid

Hamiltonian circuit of the grid). Thus, there will be no time penalty associated with the simulation.

Theorem 2 *A Hamiltonian circuit of a square grid exists $\Longleftrightarrow$ the side length of the grid is even.*

Proof: (i) side length even $\Longrightarrow$ 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 4.2 illustrates the instance with side length six.

(ii) side length odd $\Longrightarrow$ 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. $\bullet$

4.3.2 Combining Objects

To complete the manipulations involved 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 from circulation. We must also be careful to combine groups of objects in a way which respects the underlying “best partner” structure.

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 4.6 that the simplicity of these approaches will lead to superiority over other methods, in all but the most trivial (and unlikely) of examples.

4.3.2.1 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 3 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). Furthermore, the scheme clearly respects the restrictions upon the ordering of combinations implied by the need to respect best partnerships. This is because combinations only take place on the basis of a direct “best partners” relationship between two objects or between two of their already subsumed members (by virtue of information in the second field of the packets). Thus, in terms of the simple minimum spanning tree example presented

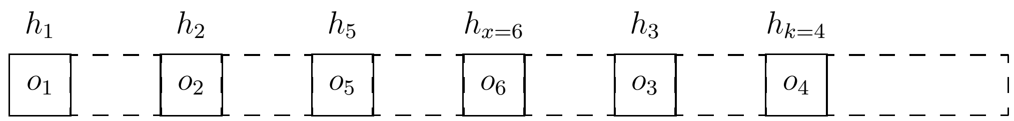

Figure 4.3: Combining object o_1 and o_6.

in section 4.1.2, there is no way that t_1 will be (incorrectly) combined directly with t_3 since packets containing these trees will only "recognise" each other if one of them also contains t_2 already.

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 3 *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. Figure 4.3 illustrates such a situation where $x = 6$ and $k = 4$.

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 (and consequently by the inductive hypothesis). This completes the proof of part 1.The proof of part 2 is entirely analogous, with the induction being from $k+1$ up to x.

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 which will be responsible for them during the next iteration.

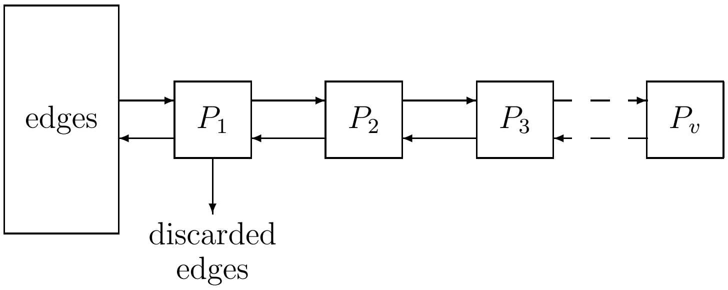

Figure 4.4: Savage's Systolic Array

4.3.2.2 The Four Tour Algorithm

This 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 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 any connected components algorithm will also solve the "lowest object identifier" problem.

Savage [31] 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 4.4, 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 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

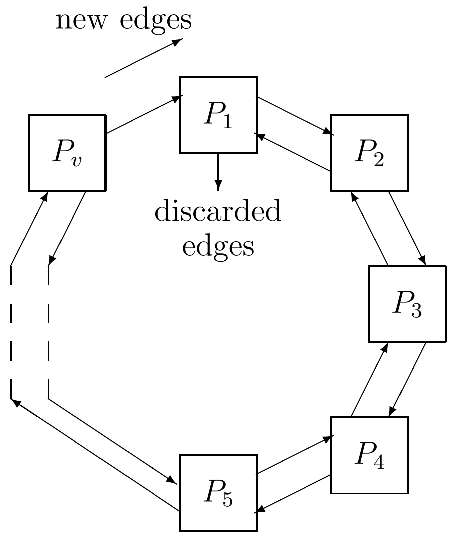

Figure 4.5: Embedding the pool in the tour

overheads[2]. 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, we note 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 the link which joins the first processor to the n^{th}, as suggested in figure 4.5. 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 to its originating processor in the "edge pool".

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 direct sequential 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

[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.

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 second algorithm requires large objects to be constructed sequentially by the processors responsible for them at the next iteration. A sensible decision as to the relative effectiveness of the two algorithms could only be made on the evidence of real examples on a real machine.

4.3.3 Checking for Termination

At this point the implementation of an 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. Otherwise, another iteration must be initiated. The sum and broadcast algorithm of section 6.2.1.3 is suitable for generating and distributing the count and takes insignificant time in comparison with the tours involved in the rest of the iteration.

4.4 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. We will consider these separately. In the first (and given the fixed size of any real machine almost inevitable) case, 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 the number of objects falls. A second possibility, considered in section 4.5, 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 sub-regions of the grid. This situation will also occur in the final iterations of instances which initially fell into the first category.

4.4.1 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 our grid machines are physically constructed this way). 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 skeleton, these examples give helpful insight into the seriousness of difficulties which may arise.

4.4.2 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 $\frac{S_0}{n}$ (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 buffer, forwarding messages between its neighbours.

4.4.3 Examples and Analysis

The scheme proposed amounts to a fixed size pipeline simulating a variably longer pipeline. It is well known [17] 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. Where do these feature in our 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 some other processors 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 between stages of the virtual pipeline, and consequently to bottlenecks.

We now present a summary of some examples 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 sizes 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 represents a possible pattern of combinations for a complete execution, with different examples varying the speed with which the number of objects falls (thereby varying the number of iterations) and the way in which discrepancy in size between remaining objects occurs (thereby varying the time taken for particular iterations). The best and worst possible scenarios (in terms of the balance of object distribution at each iteration) are compared. The best scenario models the situation in which all remaining objects at each iteration are evenly balanced (in number) 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, which would ideally be able to turn potentially worst case distributions into best cases. Of course, to be useful, such an algorithm would have to take less time than the saving produced!

The characteristics of the selected examples are presented below, with a summary of the best and worst case behaviour for each listed in figure 4.4.3.

We will be particularly interested in behaviour when the initial number of objects present, S_0, greatly outnumbers the number of processors available. 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. To capture this characteristic and to simplify results for convenient presentation, the results presented in figure 4.4.3 refer to situations in which $S_0 = \Omega\left(n^2\right)$.

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 "dramatically" 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 stay at 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.

Example 2

As in example 1 except that the first dramatic iteration in which the number of objects fell from S_0 to $\frac{S_0}{n}$, has been replaced by a more gradual process, over $\log_2 n$ iterations, with the number of objects falling by half in each of these. After this, execution proceeds as in example 1.

Example 3

The number of objects falls as in example 1. However, the combinations occur more evenly - all objects gradually increase in size and no single object becomes much larger than all the others.

Example 4

Completing the logical quartet of examples, the number of objects falls as in example 2, but without the build-up of an anomalously large object.

Examples 1-4 deal with instances in which the number of objects is halved between iterations (with the possible exception of the first iteration). The next example considers an instance in which the number of objects present falls much less rapidly.

Example 5

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. 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.

In Examples 1-5 it was assumed that initial set of objects was evenly distributed between the real processors. The final 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. It also turns out that the analyses for these examples are the same whether or not we introduce a "dramatic" first iteration. Thus the only variable factor left is the presence or absence of an anomalously large object.

Example	Best Case Time	Worst Case Time
1	$O\left(\frac{{S_0}^3}{n^3} + S_0 n \log n\right)$	$\Omega\left(\frac{{S_0}^3}{n^2}\right)$
2	$O\left(\frac{{S_0}^3}{n^3} + S_0 n \log n\right)$	$\Omega\left(\frac{{S_0}^3}{n^2}\right)$
3	$O\left(\frac{{S_0}^2}{n} + S_0 n \log n\right)$	$\Omega\left(S_0 n\right)$
4	$O\left(\frac{{S_0}^2}{n} + S_0 n \log n\right)$	$\Omega\left(\frac{{S_0}^2}{n} \log n\right)$
5	$O\left(\frac{{S_0}^4}{n^3}\right)$	$\Omega\left(\frac{{S_0}^4}{n^3}\right)$
6	$\Theta\left(\frac{{S_0}^3}{n^2}\right)$	
7	$\Theta\left({S_0}^2\right)$	

Figure 4.6: Summary of Results

Example 6

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 (i.e. there is one).

Example 7

As in example 6, except that the combinations are assumed to be more even, with no single large object appearing.

4.4.4 Summary and Discussion of Examples

Any conclusions drawn from the preceding analyses must be prefaced by two qualifications, noting the relatively small number of examples and the dangers of placing too much emphasis on asymptotic 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 which should have a bearing on implementation decisions relating to the distribution of objects.

In this spirit, the following points are noted :

- a well balanced initial distribution of objects is essential (consider example 6 and compare with the best cases of examples 1 and 2, or example 7 and compare with the best cases of examples 3 and 4);
- without the appearance of large objects redistribution cannot guarantee to offer dramatic savings (consider the difference between best and worst cases in examples 3 and 4)[3];

[3] In example 3, the performance appears to be so similar that our analysis couldn't force any gap between best and worst cases.

- the slower the rate of appearance of large objects, the less effect they have (compare the difference between best and worst cases of example 5 with those of examples 1 and 2), presumably since more work is done in the early stages when objects are still reasonably distributed;

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.

Detailed analysis reveals that 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 - the presence of one large object on a processor introduces a bottleneck into the pipeline of virtual processors simulated by that processor, to the extent that all other objects present there might as well be equally large.

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.

4.4.5 Implementing Redistribution

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

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 have been 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 will only have the effect of increasing the total time by a factor of ≤ 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.

4.5 Redistribution with a Shortage of Objects

The issue of redistribution can appear in a complementary way to that discussed above. Instead of a having 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 (i.e. there will be a gap in the pipeline). 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. This is the approach we will take. 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 every time the number of objects falls to half the number of currently active processors.

4.5.1 The Goal

A crucial iteration is one 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 positive 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 4.7.

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 to become idle. These "free" objects must be relocated to processors within the inner grid which currently have no home object. Such free 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 (or neither) as a direct consequence of its absolute location in the 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 $\Theta(k \log k)$. 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.

4.5.2 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. Thus, any choice of location for the new sub-grid would be handled equally well.

The essence of the algorithm is to repeatedly partition the $k \times k$ grid into smaller and smaller pieces, while correcting the distribution of free objects be-

[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, which seems plausible. It places no restrictions on the use of the skeleton.

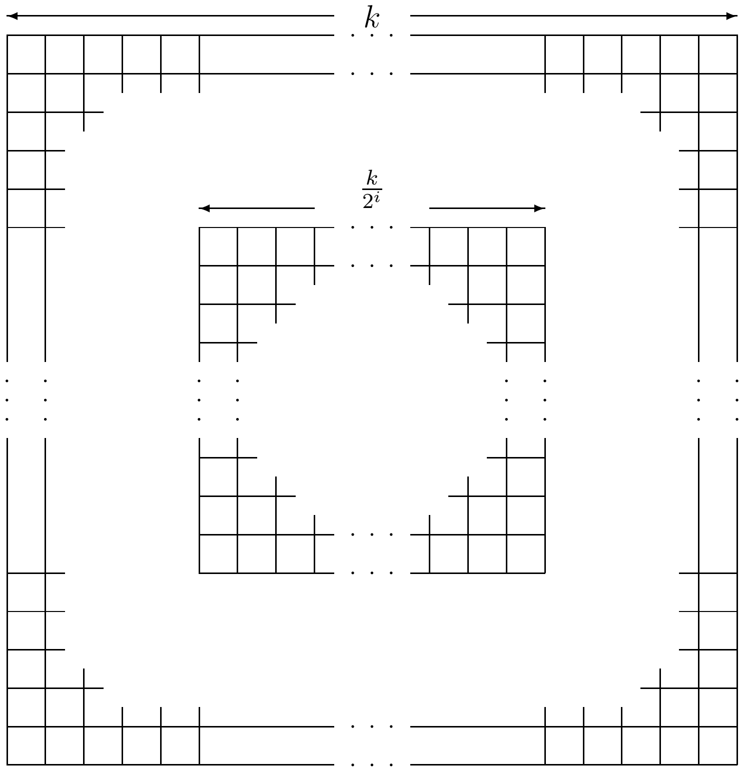

Figure 4.7: Redistributing to a smaller grid

tween 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, at which point overall redistribution is complete.

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 $\frac{k^2}{2^{i-1}}$ 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 redistribution process would not have been instigated), it may be

[5]By partitioning a square into two equal rectangles or a rectangle into two equal squares, as i is odd or even respectively.

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 such occurrences at each level of partition, by moving an appropriate number of free objects from the overloaded group to its "partner".

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 6.2.1.3 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. As we have seen, at most one member of any newly created pair of groups will be an "exporter".

2. Processors in an exporting square determine precisely which $f_* - h_*$ objects will be sent across the boundary to their partner group. 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. An object originating in the i^{th} row and j^{th} column of the exporting square moves to the corresponding processor in the importing square. Since the whole shunt is effectively pipelined across rows (or down columns), this operation takes $k-1$ time steps between squares of side k. The 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 4 *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 $\frac{k}{2^{i-1}}$, if i is even and $\frac{k}{2^i}$ 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 takes $\Theta(k \log k)$ time. •

4.5.3 Evaluating the Algorithm

The examples presented in section 4.4.3 , dealing with the performance of the proposed redistribution algorithm for $|S| >> n$, 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 $|S_0| >> n$), 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. However, our analysis has shown that the scheme is to within a factor of $O(\log n)$ of optimal for any such algorithm, with its $O(\sqrt{n} \log \sqrt{n})$ execution time. Bearing in mind the substantial discrepancies in performance which could be avoided by redistribution in the situations considered previously, it seems quite likely that similar savings could be made here.

4.6 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 familiar problems of contention for links and the resulting bottlenecks would have to be considered, especially for cases in which most objects were 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 dynamic 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 polls were initiated before the end of the phase actually occurred. These considerations tentatively 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.

4.7 Examples

4.7.1 Minimum Spanning Tree and Connected Components

Sollin's algorithm [33] to find a minimum weight spanning tree of a weighted graph was presented in section 4.1.1 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 "combine", which combines two such trees by their shortest joining edge and returns a description of the resulting tree, and a function "value", which returns the weight of the joining edge. If no such edge exists then "value" can return "infinitely" high weight,

- a function "accept" which compares the costs returned by two possible combinations, returning "true" if the first is preferable to the second and "false" otherwise.

- 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 ∞ 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

$$v\left(v^2+\left(\frac{v}{2}\right)^2+\ldots+2^2\right)=\Theta\left(v^3\right).$$

Analysis of the parallel implementation follows a pattern similar to that presented in example 4 of section 4.4.3. For the MST example it must be noted that the size of "objects" is now $\Theta(v)$ throughout (because of the array). The overall time taken turns out to be $\Theta\left(\frac{v^3}{n}+v^2 n\log n\right)$ 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 $\Theta(v^2 \log v)$. Unfortunately, this improvement has no effect on the skeleton implementation. It is still necessary to move objects of size $\Theta(v)$ 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.

4.7.2 Partitioning Programmable Logic Arrays

The programmable logic array (PLA) 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 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 $\Theta\left(\frac{n}{\log n}\right)$ growing to $\Theta(n)$ fold speed-up for n processors, when the problem is suitably large. Most importantly, for the PLA problem 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. This is because 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 in the spanning tree example. PLA partitioning provides an example of an application which can exploit the iterative combination skeleton with high efficiency.

Chapter 5

The Cluster Skeleton

5.1 An Alternative Approach to Skeleton Construction

The skeletons discussed in chapters 3,4 and 6 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 who is responsible for extracting 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 pattern of decomposition and distribution for which the structure of the grid seems especially suited. Then, from the bottom up, we construct an abstract specification with its implementation based on this pattern.

5.2 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 that it doesn't appear to 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" [6]). 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 on the observation, we note that theorem 2 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.

5.2.1 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 the two skeletons discussed previously. Specification of the FDDC 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 certain operations. We now expand upon these observations to specify a new skeleton.

The FDDC skeleton required its problems to be split into some specified number of sub-problems. This division of problems and the subsequent combination of results was initiated centrally before spreading across the machine. The unrestricted nature of the division of the grid now under consideration, suggests that the requirement that sub-problem generation be restricted to a specific constant should be discarded. Equally, the co-operative style of solution promoted by the use of Hamiltonian circuits seems to point away from the centralised division and combination found in the FDDC 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.

Considered together, these ideas 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. More precisely, 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 will be considered together with all other members of their parent cluster and operated upon in some way with respect to each of 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 "closeness" 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.

5.2.2 Cluster Skeleton 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.

Suppose that the data objects which describe a problem instance are of type "*obj*" and that the level of recursion in the cluster hierarchy is of type "*depth*". To produce a problem specific "Cluster" program, the programmer must specify three functions

$$\begin{array}{ll} match & : depth \rightarrow obj \rightarrow obj \rightarrow bool \\ reshape & : depth \rightarrow obj \rightarrow obj \rightarrow obj \\ continue & : depth \rightarrow obj \rightarrow bool \end{array}$$

where *match* determines whether two objects should be clustered together at the given depth, *reshape* describes how to update the first object to take account of the presence of the second in its cluster at the given depth and *continue* decides whether recursion should proceed for a particular cluster beyond the current depth by inspecting any of that cluster's members.

Given these definitions, the system constructs two new functions

$$\begin{array}{ll} decompose & : depth \rightarrow \{obj\} \rightarrow \{\{obj\}\} \\ amend & : depth \rightarrow \{obj\} \rightarrow \{obj\} \end{array}$$

such that *decompose* applies *match* (parameterized by the current depth) to every pair of objects in the cluster, returning the set of clusters implied by the resulting partition. Similarly, *amend* applies *reshape* to every pair of objects in its argument cluster.

Assuming that "*union*" is a function which returns a cluster representing the union of a set of clusters

$$union : \{\{obj\}\} \rightarrow \{obj\}$$

and that "*map*" operates over sets as it does over lists, then a full specification of the cluster skeleton as a higher order function would have the form

$$\begin{array}{rl} cluster\ match\ reshape\ continue = & C\ 0 \\ \text{where } C\ d\ set = & amend\ d\ union((map\ (C\ d{+}1)\ (decompose\ d\ set)), \\ & \text{if } continue\ d{+}1\ s \\ = & amend\ d\ set, \text{ otherwise} \end{array}$$

where s represents any element of *set*. Thus, the skeleton has type

$$\begin{array}{ll} cluster: & (depth \rightarrow obj \rightarrow obj \rightarrow bool) \rightarrow (depth \rightarrow obj \rightarrow obj \rightarrow obj) \rightarrow \\ & (depth \rightarrow obj \rightarrow bool) \rightarrow \{obj\} \rightarrow \{obj\} \end{array}$$

5.3 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.

5.3.1 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 involved 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 sub-clusters at the next level of recursion, with a subset of the original group of processors 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 work required by "*amend*".

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 efficiently implement the "all possible pairs" pattern required by both "*decompose*" and "*amend*".

5.3.2 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 functions

These conditions are obviously satisfiable at the level 0, when the "cluster" contains all the objects and the "group" encompasses the whole grid. Section 5.3.3 will show how they may be maintained from one level to the next.

Execution begins with evaluation of the function "*continue*". Since we have already required that it should be decidable upon examination of any 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 apply "*match*" to all pairs of objects in the cluster 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 "*match l s t*" with its own 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 4.3.2.2 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 5.3.3 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 5.3.3 ensure that these are still members of the same cluster. Thus, "*union*" is implemented at no cost by simply forgetting about the lower level cluster structure from which the objects were returned.

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

5.3.3 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 initial situation is that 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.

The presentation is divided into two sections. In the first we describe an idealised implementation which assumes the processors to be connected into two chains of equal length with additional links between the chains, connecting corresponding processors. In the second section, we show how this structure can be conveniently embedded into the grid with only constant overheads, thereby achieving an efficient practical implementation.

5.3.3.1 An Idealised Solution

In the idealised solution we will make two extra assumptions, firstly that p is even and secondly 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_{p-i+1}. These additional links (which give the "chain of pairs" effect) are shown as vertical dotted lines in figure 5.1 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.

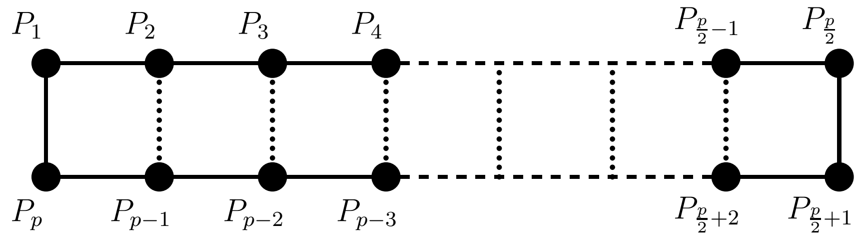

Figure 5.1: Processor Identifiers and Links

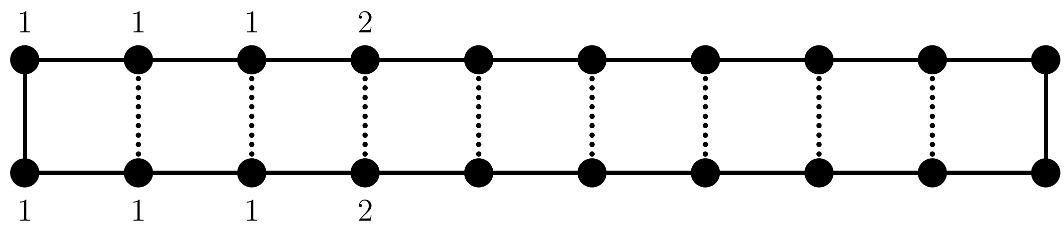

Figure 5.2: Allocation of the first two clusters

Suppose that the c objects are to be partitioned into k sub-clusters of sizes $c_1, c_2, ..., c_k$. Then the best possible reorganisation would assign these to groups of $\frac{c_1}{c}p, \frac{c_2}{c}p, ..., \frac{c_k}{c}p$ processors respectively (to the nearest integers). Our solution will approximate this by assigning to groups of $d_1, d_2, ..., d_k$ processors, where d_i is the largest even number no larger than $\frac{c_i}{c}p$ when $\frac{c_i}{c}p$ is at least 2, and 1 when $\frac{c_i}{c}p$ 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 $\frac{c_i}{c}p < 2$) clusters are assigned to groups of processors from left to right along figure 5.1, 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 5.2.

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.

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 5.3.

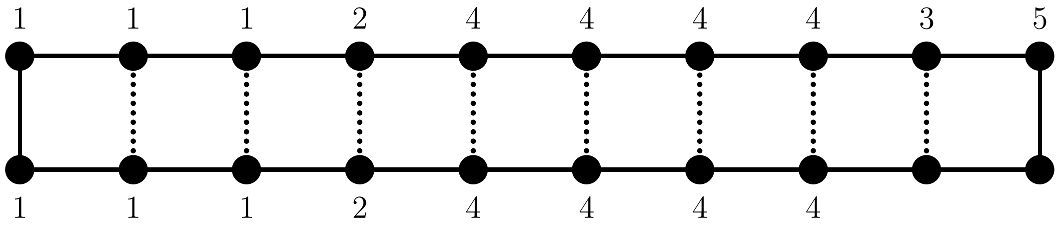

Figure 5.3: The complete allocation

There are two important points to note about the allocation procedure. Firstly, it is completely deterministic for given $c, c_1, ..., c_k$ 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 5.4, allowing the whole procedure to be repeated 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

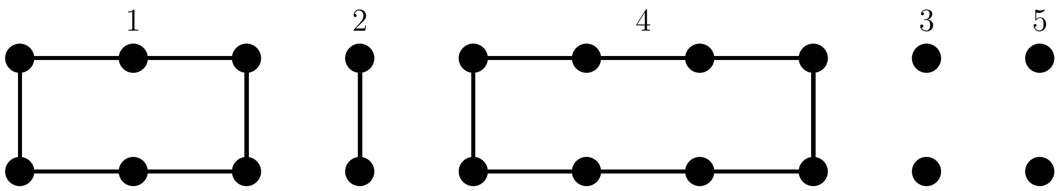

Figure 5.4: Circuits at the next level

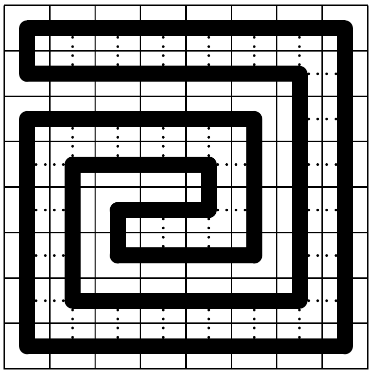

Figure 5.5: A useful Hamiltonian circuit

new objects (the choice between particular processors in the group for particular objects within a cluster is irrelevant).

5.3.3.2 Practicalities

The idealised solution above assumed that the links shown as dotted lines in figure 5.1 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 5.5 comes close enough to make a simple adaptation of the abstract solution practical. Almost all of the extra links required by our idealised solution are present as physical links, as illustrated. 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 5.6 illustrates such a situation. It is unsuitable to break the circuit portion of part (a) as illustrated in part (b), since this would leave processors x and x+1

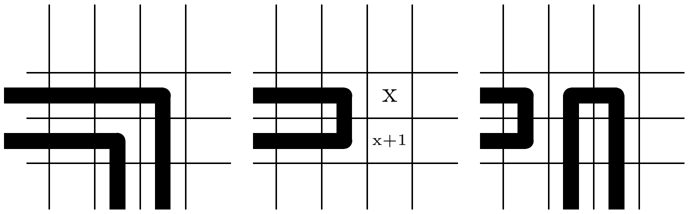

Figure 5.6: An illegal break

with no suitable partners, and therefore wasted[1]. 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.

5.4 Exploiting the Cluster Skeleton

We have presented the cluster skeleton as an example of an experiment in skeleton design "from the machine out" rather than "from the algorithms in". It is not difficult to see that the implementation proposed is very efficient. 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.

Unfortunately, the evolution of the skeleton's definition and in particular its lack of a pool of motivating algorithms means that we have no immediate selection of existing problems from which to draw examples. The true utility of the skeleton

[1] 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.

will only emerge when it can be presented as a real tool. However, in order to give the feel of the type of situation in which it might prove appropriate, we now sketch a possible application.

Consider a problem which may occur when analysing some large body of data describing some scene or event. For example these may be digitised portions of some overall image. There may be a large number of such observations, originating from a variety of viewpoints 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 describe the same real object and to coalesce each such group into a clearer and hopefully more accurate description, eventually distilling a coherent picture of the whole scene.

A wide variety of techniques may be used in comparing and coalescing such images, with varying degrees of complexity. In deciding which portions of data 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 satisfactorily in most cases. A suitable approach to the problem could be expressed in terms of the cluster skeleton. 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 while maintaining their own observer location information and so on.

Chapter 6

The Task Queue Skeleton

Our final skeleton differs most significantly from the other three in having an explicitly parallel specification. It is interesting to note that the isolation of individual skeletons within the overall framework means that parallelism can be introduced at this level in a restricted way, as appropriate. There are no interactions with other constructs to be considered, either for programmer or implementor, as there would be in a conventional "universal" language. Section 6.1 motivates and introduces this specification. Section 6.2 introduces the overall structure of a proposed implementation which is discussed in detail in sections 6.3 and 6.4 and summarised in section 6.5. Finally, section 6.6 presents some problems for which solutions may be derived from this "task queue" skeleton.

6.1 The Abstract Specification

6.1.1 Motivation

The class of algorithms from which the task queue skeleton has been distilled 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 many instances 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 an example.

The one-to-all shortest paths 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. [30] present a parallel algorithm which solves this problem for graphs containing no negative cycles (otherwise 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 ∞ 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 waiting tasks. The task queue is initialized to contain a single task parameter, the identifier of the selected central node, to which a "shortest path" 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 on average, 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 corresponding instantiations of the task procedure. The task queue skeleton has a similar feel to the "ask-for" monitor of [5].

6.1.2 Specification

A more precise abstract specification of the task queue skeleton can now be presented. Although we have now introduced explicit parallelism, we still wish to maintain a high degree of machine independence. Thus, the user must consider the skeleton to be executed on an unspecified number of parallel processors, each independently executing the loop shown in figure 6.1. 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 example, 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.

```
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 6.1: Idealised processor code

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, but independently in terms of operations performed. 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 imaginary 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 emphasise 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, as circumstances dictate (consider using a first-in first-out discipline, for example). 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 different tasks. For different numbers of processors, a certain instruction from one task may or may not be executed before some other instruction 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. As a result, the concept of clashing accesses has no meaning at the user's abstract level, since its occurrence can never be determined without knowledge of the implementation (which, of course, should not be available). 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 6.6 suggest that it poses no practical difficulties.

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 evolving 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 procedure 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.

6.2 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 in out target hardware 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 local instructions. Clearly execution of instructions of the first two types may involve references to objects stored non-locally thereby requiring 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 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 6.3 discusses algorithms which could be used to implement the two phases involving access to the task descriptor queue, while section 6.4 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

[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.

as building blocks.

6.2.1 Some Useful Algorithms

6.2.1.1 Sorting

Theorem 5 *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.*

Proof:

One such algorithm is presented by Thompson & Kung [34]. •

6.2.1.2 Routing

Theorem 6 *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 described by Valiant & Brebner [37]. 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 by inspecting 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 6.2. 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 synchronisation 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 waiting to leave a processor on a particular link. No such queue can be encountered when

```
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 6.2: A Routing Algorithm

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. •

6.2.1.2.1 Augmented Routing

Theorem 7 *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. •

6.2.1.3 Sum & Broadcast

Theorem 8 *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 $4\left(\sqrt{n}-1\right)$ 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. •

6.3 Implementing the Queue

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

6.3.1 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 stack (in arbitrary order amongst themselves) and the r requests are then served from the new head of the stack backwards. Thus if $r \leq p$ then the requests will all be served with tasks just placed on the stack. 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. We will refer to processors which require to interact with the stack at a particular idealised time step as being "active".

Implementation requires some algorithm which can effect this idealised behaviour. This algorithm will be executed at every idealised step and its run time will contribute to 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. Its performance is shown to be asymptotically optimal for the n processor grid.

6.3.1.1 Implementing the Stack on the Grid

A simple way to implement the stack of available tasks would be to make one processor responsible for its contents and for the handling of any accesses to it required by active processors. Unfortunately, the performance of such a centralised approach would be very vulnerable to idealised time steps in which many processors are active, since their requests would have to be serviced sequentially. In contrast, our proposed implementation is based upon a grid-wide distribution of the stack, in which the task descriptors involved in any particular idealised time step will be handled concurrently by distinct processors.

Given such a distribution, there are two remaining problems. Firstly we must describe the means by which an active processor can determine the the processor with which it should interact to access an appropriate element of the stack. Secondly we must show how a set of such interactions can proceed concurrently with good performance.

6.3.1.2 Distributing the Stack

The task stack contains a number of descriptors which varies over time. Our goal is to define a mapping of these descriptors to the grid such that at any time, any active processor can easily determine the location of any particular "stack" element. The solution proposed here is very simple. Firstly, we assume the existence of some pre-defined ordering on processors, known to all in advance[2]. The first processor in the ordering is nominated as the initial location of the "stack head" and is allocated the descriptor of the top task on the initial stack. Successive descriptors are allocated to successive processors in the underlying ordering, with wrap around from last to first. Thus the i^{th} processor's memory will initially contain the i^{th}, $(i+n)^{th}$, etc. descriptors, and any processor can immediately determine the location of the j^{th} descriptor provided that it knows the location of the "stack head".

As execution proceeds, the size of the stack will vary. Our mapping allows us to cater for this conveniently. Suppose that p new tasks are to be added to the stack. Our mapping assigns these to the p processors preceding the current "stack head" processor in the underlying ordering. Then, every processor is made aware that the stack head has now moved from its previous location to the processor p places away in the underlying ordering. Similarly if r tasks are to be removed from the stack, then these must be taken from the current "stack head" processor and its $r-1$ successors in the ordering. Again, all processors must be made aware that the "stack head" has moved.

[2]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.

6.3.1.3 Managing the Stack

The algorithm required to manage stack accesses (for any particular idealised time step) has two distinct phases which deal with addition and removal of tasks respectively. The overall structures of the two phases are almost identical. In the first, each of the p processors requiring to add new tasks to the stack must determine which processor will be responsible for storing the newly generated descriptor. It then sends the descriptor to this processor. Finally all processors are made aware of the new location of the stack head. The second phase proceeds similarly, with processors requiring new tasks identifying and interacting with the processors storing these tasks.

In both cases, the success of the algorithm depends upon ensuring that each active processor identifies a distinct "stack location" processor with which to interact. For example, in the first phase, exactly one active processor should send its new descriptor to processor "stack head - 1", exactly one to "stack head - 2", and so on. Thus, the problem boils down to being able to make each active processor generate a unique "offset" in the range 1 to p (or 1 to r in the second phase). We can note immediately that any such generation will do, since the abstract specification gives no particular active processor the right to any particular element within the appropriate range. Therefore, we can safely make the arbitrary decision that offsets 1 to p will be allocated to processors in ascending order of their position in the underlying ordering on processors. For example, if processor 1 in the ordering is active, then it will always generate an offset of 1. Theorem 9 proves that this scheme has the desirable property that for any particular idealised phase, no two active processors will ever try to interact with the same "stack" processor.

Theorem 9 *Each processor will be responsible for at most one accessed stack location in each phase.*

Proof : The locations have been distributed cyclically across all n 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. •

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 6.3: Row-major ordering

0	0	0	1
0	0	1	0
0	1	0	0
0	0	1	1

Figure 6.4: Active processors.

-	-	-	1
-	-	2	-
-	3	-	-
-	-	4	5

Figure 6.5: Appropriate offsets.

6.3.1.4 Generating Offsets

We assume that the underlying ordering of processors assigns identifiers in "row-major" order, as illustrated in figure 6.3. At the beginning of each phase, every processor generates an initial offset value of 1 if it is to be active during the phase, or 0 otherwise. From this position, an appropriate offset for each active processor is then the sum of the offset values in all processors with smaller identifiers (in the underlying ordering) than its own, together with its own offset of 1.

For example, if the active processors are as indicated in figure 6.4, then appropriate set of offsets is shown in figure 6.5. Thus, to calculate its own unique offset, an active processor must sum the offsets generated to its left in its own row, together with its own offset and the offsets of all processors in the rows above. This summation can be achieved in $O(\sqrt{n})$ time with a simple adaptation of the two stage sum and broadcast algorithm of section 6.2.1.3.

In the first phase, a single packet is created in the leftmost processor of each row. This contains the initial offset (i.e. 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.

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 $\Theta(\sqrt{n})$.

The algorithm has the side effect that processor $n-1$ becomes aware of the total number of active processors, irrespective of its own state. Since this is the number by which the "stack head" has moved, it can be broadcast to all processors at the end of the phase, enabling them to update their copy of the "stack head" location appropriately.

6.3.1.5 Accessing the Queue

Suppose the current phase involves adding tasks to the stack. Then, with each active processor having located its target processor, 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 9, so are all destinations. Thus, the algorithm of section 6.2.1.2 can be used to achieve the routing in no more than $3\sqrt{n}-3$ moves. Finally, the new "stack 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 6.2.1.3.

As before, the procedure for simulating a phase in which elements are removed from the stack 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 $\Omega\left(\sqrt{n}\right)$ 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.*

Proof : Consider two consecutive idealised time steps. In the first, exactly one processor writes a new task to the stack 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 $\Theta\left(\sqrt{n}\right)$ apart and will thus require $\Omega\left(\sqrt{n}\right)$ time simply to route the task across the grid. •

6.3.2 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.

6.3.3 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 $\Theta\left(\sqrt{n}\right)$ links away in the next step. Thus, if it must be ensured that every request is served before computation may proceed, a $\Theta\left(\sqrt{n}\right)$ 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 once again be used to implement this specification of the queue.

6.3.4 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 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 6.2.1.1 as a subroutine. 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 6.3.1. 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 6.2.1.3. 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 4.5.2) is used to balance the queue elements evenly between processors. Since each processor can have at most one surplus element, a single run of this $\Theta(\sqrt{n}\log(\sqrt{n}))$

algorithm 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 $O\left(\sqrt{n}\log\left(\sqrt{n}\right)\right)$ 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 6.2.1.1 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 $O\left(\sqrt{kn}\right)$ which is just $O\left(\sqrt{n}\right)$ 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 6.2.1.2. 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 $O\left(\sqrt{n}\right)$.

The whole algorithm executes in $\Theta\left(\sqrt{n}\log\left(\sqrt{n}\right)\right)$ 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.

6.3.5 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 6.5. Since the simulation already includes $\Theta(\sqrt{n})$ slow down, such an addition would only increase overall run time by a constant factor.

6.4 Implementing the Data Structure

In this section, some solutions to the problems surrounding the shared data structure component of the skeleton are considered. Section 6.1 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. 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. 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 worst case $\Theta(n)$ time slow down. Fortunately, we can exploit a feature of the skeleton's specification to avoid this problem.

In a conventional shared memory simulation (e.g. [37]) 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 predict 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 unpredictable 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. 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 an issue. 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 low order interleaving scheme for the virtual to physical memory mapping, resulting in adjacent virtual addresses being located in independent physical memories.

Section 6.4.1 now considers the structure of a suitable implementation algorithm and properties required of it, while section 6.4.2 presents some variations on its internal details.

6.4.1 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 locations, or even unique processors

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. In the approach presented here, 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 precise 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 – accesses which were successful, but unacknowledged, 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 stronger property can be guaranteed without significantly affecting the time needed for the second phase, t_2.

6.4.2 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[3]. This would guarantee that the second phase (sending acknowledgements) involved unique sources and unique destinations. Using the routing algorithm of section 6.2.1.2, 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.

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 10 *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 11 *Use of the standard routing algorithm in the first phase does maintain property 1.*

[3] In 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.

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 10 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 12 *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 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 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.

6.5 Summary

Sections 6.2, 6.3 and 6.4 discuss grid implementation of the abstract task queue skeleton specified in section 6.1. 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.

6.6 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 general estimates of performance impossible. 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.

6.6.1 One to All Shortest Paths

This may be implemented directly from the algorithm presented in section 6.1, with one alteration. The idealistic model of Deo's algorithm 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 (i.e. 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 for the "central vertex" will be written when the appropriate neighbour is inspected in this "extra" task instance.

With this addition, the algorithm presented in [30] 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.

6.6.2 LU Matrix Decomposition

A set of n linear equations in $x_1, ..., x_n$ may be described by the equation

$$Ax = b$$

where A is an $n \times n$ coefficient matrix, x is the vector $(x_1, ..., x_n)$ 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 [38] for a thorough discussion). Overall solution is then simplified to the task of solving

$$Ly = b$$

to obtain an intermediate vector y, then solving

$$Ux = y$$

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 re-calculation) 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 [38]. The first $r-1$ equations of figure 6.6 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}.

$$
\begin{array}{lllllllll}
l_{r1}u_{11} & & & & & & & = & a_{r1} \\
l_{r1}u_{12} & + & l_{r2}u_{22} & & & & & = & a_{r2} \\
 & . & & . & & . & & & . \\
l_{r1}u_{1r} & + & l_{r2}u_{2r} & + & ... & + & l_{rr}u_{rr} & = & a_{rr} \\
l_{r1}u_{1,r+1} & + & l_{r2}u_{2,r+1} & + & ... & + & l_{rr}u_{r,r+1} & = & a_{r,r+1} \\
 & . & & . & & . & & & . \\
l_{r1}u_{1n} & + & l_{r2}u_{2n} & + & ... & + & l_{rr}u_{rn} & = & a_{rn}
\end{array}
$$

Figure 6.6: Producing the r^{th} row of L and U

Similarly, the remaining $n-r$ equations specify $u_{rr}, ..., u_{rn}$ 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 $u_{1i} = a_{1i}, (1 \leq i \leq n)$ 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 6.7.

```
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+1];  {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;

  IF c=r-1 THEN BEGIN

    WHILE mark[r,i] <> r-1 DO ; {wait for completion of row r}

    IF r+1 < n THEN put_task(r+1,1); {start generation of row r+1}

  END;

END;
```

Figure 6.7: Code for the LU decomposition task

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.

6.6.3 Solving a Band Matrix System of Linear Equations

McKeown [23] 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 b elements in the current solution). Maximum concurrency is bounded by b, but synchronisation of the kind encountered in the preceding example is not required.

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 a programming model which allows clear and easy specification of solutions to their problems. In particular, the model should be independent of the details of the underlying parallel hardware. Traditional problems 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?" wonders 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 principle characteristics of a spectrum of systems which address the problem. This led, in chapter 2, to the proposal of a new style of abstraction, based on the notion of algorithmic skeletons. Subsequent chapters centred on the specification and implementation of one such system, consisting 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 in the first chapter. 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 programming model. 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 is 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. More precisely, 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. Since the four skeletons presented are in no way assumed to be exhaustive, it is to be hoped that this will not occur frequently with a full system. Future studies of existing algorithms and attempts to use the model 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 (or declarative, according to taste) 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 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.

A final yardstick against which we may assess the skeletal machine is the popular notion of the "grain" of a parallel machine or algorithm. Essentially, this is a rough measure of the ratio of computational steps to communications to which a machine is best suited or to which an algorithm is most prone. A "coarse grain" indicates that this ratio is high, in contrast to "fine grain" at the other extreme. To a certain extent the skeletal machine in its abstract form is independent of this notion - the "grain" is provided by the programmer who specifies the problem specific functions. The resulting algorithm is coarse or fine just as these are. However, with an eye to good practical performance, it seems that the implementation overhead constants which have disappeared into our asymptotic analyses would be best hidden by "coarse" customising functions. This will be especially true while the relative costs of real computation and communication remain as unbalanced (in favour of computation) as they are at present. Certainly, the complexity of the hardware which we would require at each node to implement the skeletons is coarse grain by today's standards. However, the implementation algorithms proposed here have a "systolic" feel which might be well suited to direct realisation in hardware at some point in the future.

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 [13] 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 $\Theta(\log n)$, or a small function thereof.

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 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 6. Global and local coalescence and broadcasting of information appeared in several contexts. The implementations presented in chapters 4 and 5 were founded upon the existence of Hamiltonian circuits linking all processors.

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 require investigation would be conveniently partitioned for concurrent consideration!

Bibliography

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullmann. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. (As useful as ever).

[2] G. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin/Cummings, 1989. (A thorough and entertaining survey, with an excellent bibliography).

[3] Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. *Journal of Parallel and Distributed Computing*, 5:460–493, 1988. (Describes the ID Nouveau language and its advantages over Fortran for parallel scientific programming).

[4] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988. (An excellent introduction to the topic).

[5] J. Boyle, R. Butler, T. Disz, B. Glickfield, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Computers*. Holt, Rinehart and Winston, 1987. (The design an implementation of a collection of machine independent communication and synchronisation aids for the explicitly parallel programmer).

[6] G.J. Brebner. Relating Routing Graphs and Two Dimensional Grids. In Bertolazzi and Luccio, editors, *VLSI Algorithms and Architectures*, North-Holland, 1985. (The problems involved in laying out good interconnection networks in VLSI).

[7] J. Darlington and M. Reeve. A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages. In *ACM Conference on Functional Programming Languages and Computer Architecture*, pages 65–74, 1981. (An early description of ALICE).

[8] J. Darlington and M. Reeve. Alice and the Parallel Evaluation of Logic Programs. In *10th Annual Symposium on Computer Architecture*, 1983. (Empha-

sises the connections between the parallel evaluation of logic and functional languages).

[9] J.B. Dennis. Data Flow Supercomputers. *Computer*, 48–56, November 1980. (A good introduction to Data Flow architectures).

[10] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988. (A broad selection of algorithms for PRAMs).

[11] S. Gorn. Letter to the Editor. *CACM*, 1(1):2–4, 1958. (Early recognition of the benefits and difficulties of parallel programming).

[12] A. Gottlieb, R. Grishman, C.P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer – Designing an MIMD Shared Memory Parallel Computer. *IEEE TC*, C-32(2):175–189, 1983. (Concentrates on architectural features and describes the implementation of fetch and add).

[13] L.J. Guibas, H.T. Kung, and C.D. Thompson. Direct Implementation of Combinatorial Algorithms. In *Proceedings of the Caltech Conference on VLSI*, pages 509–525, 1979. (Dynamic programming on a systolic grid).

[14] E. Horowitz and A. Zorat. Divide-and-Conquer for Parallel Processing. *IEEE Transactions on Computers*, TC-32(6):582–585, 1983. (Discusses direct implementation on physical trees and on bus based shared memory systems).

[15] P. Hudak. Para-Functional Programming. *IEEE Computer*, 60–70, August 1986. (The ParAlfl language and its source level facilities for controlling the execution of functional programs on multi-processor architectures).

[16] P. Hudak and B. Goldberg. Distributed Execution of Functional Programs using Serial Combinators. *IEEE Transactions on Computers*, C-34(10):881–891, 1985. (Parallel execution on non-shared memory architectures, using serial combinators to try and get the grain right).

[17] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1985. (Thorough coverage of parallel architectural principles. A bit thin on language issues).

[18] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall International, 1988. (The latest incarnation of Occam).

[19] Inmos Ltd. *The Transputer Reference Manual.* Prentice-Hall International, 1988. (A computer on a chip).

[20] P. Kelly. *Functional Programming for Loosely Coupled Multiprocessors*. Pitman, 1989. (Introduces the language Caliban, in which the partitioning and distribution of a functional program can be described in a powerful, pure, declarative style).

[21] D.E. Knuth. Big Omicron, Big Omega and Big Theta. *SIGACT News*, 18–24, April–June, 1976. (The notation of asymptotic analysis).

[22] D. L. McBurney and M.R. Sleep. *Experiments with the ZAPP: Matrix Multiply on 32 Transputers, Heuristic Search on 12 Transputers*. School of Information Systems Internal Report SYS–C87–10, University of East Anglia, 1987. (The title speaks for itself. Also explains the single steal execution rule).

[23] G. P. McKeown. A Special Purpose MIMD Parallel Processor. *Information Processing Letters*, 20:23–27, 1985. (Solving band matrix linear equations on a chordal ring architecture).

[24] G.P. McKeown, V.J. Rayward-Smith, and F.W. Burton. Towards the Parallel Implementation of a Generalised Combinatorial Search Algorithm on MIMD Parallel Computers. In C. Jesshope, editor, *Proceedings of CONPAR88, UMIST, UK*, Cambridge University Press, 1989. (On capturing the general structure of combinatorial search algorithms).

[25] Meiko Ltd. *Computing Surface Technical Specifications*. Meiko Ltd, Bristol U.K., 1987. (One way to deal with lots of transputers).

[26] K. Melhorn and U. Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984. (Introduces the idea of using multiple copies of shared memory locations in distributed simulations).

[27] P. Moller-Nielsen and J. Staunstrup. Problem-heap: A Paradigm for Multiprocessor Algorithms. *Parallel Computing*, 4:63–74, 1987. (Algorithms with a divide and conquer feel).

[28] F.J. Peters. Tree Machines and Divide and Conquer Algorithms. In *Proceedings CONPAR 81, LNCS 111*, pages 25–36, Springer-Verlag, 1981. (Analysis of direct physical tree implementations).

[29] S.L. Peyton-Jones. Parallel Implementations of Functional Programming Languages. *Computer Journal*, 32(2):175–186, 1989. (A clear overview of the important issues, with pointers to contemporary approaches).

[30] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987. (An excellent survey of parallel algorithms for a wide variety of problems and architectures).

[31] C. Savage. A Systolic Data Structure Chip for Connectivity Problems. In H.T. Kung, R. Sproull, and G. Steele, editors, *VLSI Systems and Computations*, pages 296–300, Computer Science Press, 1981. (Including a connected components algorithm).

[32] R. Sedgewick. *Algorithms*. Addison-Wesley, 1983. (Source of the divide and conquer FFT algorithm).

[33] M. Sollin. An algorithm attributed to Sollin. In S. Goodman and S. Hedetniemi, editors, *Introduction to the Design and Analysis of Algorithms*, McGraw-Hill, 1977. (A sequential minimum spanning tree algorithm).

[34] C.D Thompson and H.T Kung. Sorting on a Mesh-Connected Parallel Computer. *CACM*, 20(4):263–271, 1977. (The first asymptotically optimal sorting algorithms for this architecture).

[35] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *IFIP Conference on Functional Programming and Computer Architecture*, 1985. (An introduction to Miranda).

[36] E. Upfal and A. Wigderson. How to Share Memory in a Distributed System. *JACM*, 34(1):116–127, 1987. (Shows that multiple copies of shared memory locations can be used to handle writes efficiently in distributed simulations).

[37] L.G. Valiant and G.J. Brebner. Universal Schemes for Parallel Communication. In *Symposium on the Theory of Computing*, pages 263–277, 1981. (On the use of randomisation to improve the performance of routing algorithms).

[38] J.H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965. (Source of the LU matrix decomposition algorithm).

Index