Algorithmic Skeletons as a Method of Parallel Programming

THESIS

Submitted in fulfilment of the

requirements for the Degree of

MASTER OF SCIENCE

of Rhodes University

by

Rees Collyer Watkins

October 1992

Abstract

A new style of abstraction for program development, based on the concept of algorithmic skeletons, has been proposed in the literature. The programmer is offered a variety of independent algorithmic skeletons each of which describe the structure of a particular style of algorithm. The appropriate skeleton is used by the system to mould the solution.

Parallel programs are particularly appropriate for this technique because of their complexity. This thesis investigates algorithmic skeletons as a method of hiding the complexities of parallel programming from the user, and for guiding them towards efficient solutions.

To explore this approach, this thesis describes the implementation and benchmarking of the divide and conquer and task queue paradigms as skeletons. All but one category of problem, as implemented in this thesis, scale well over eight processors. The rate of speed up tails off when there are significant communication requirements. The results show that, with some user knowledge, efficient parallel programs can be developed using this method. The evaluation explores methods for fine tuning some skeleton programs to achieve increased efficiency.

Acknowledgements

I am very grateful for the help of my supervisors, Professor Clayton and Dr Wentworth, not least of which consisted of reading draft copies of this thesis. My thanks also to the members of the Parallel Processing Group at Rhodes University.

I am also grateful for the financial support of the Foundation of Research and Development and the Rhodes Bursary Office.

Table of Contents

Figures and Tables

Chapter 1

1. Introduction

The design of a computer system is essentially a process whereby each new level of abstraction is built on those beneath it. It has generally been found that, with each new level of abstraction, a set of more useful and more appropriate resources than is currently available, can be created. This is achieved at the expense of some freedom and efficiency, but with an increase in clarity and portability.

An Algorithmic Skeleton is an organization technique which aids in the development of programs by pre-supplying part of the design requirements. In employing a skeleton the programmer[1] identifies the class of problem to be solved and uses the appropriate skeleton to mould their solution. The skeleton has a number of well defined inputs. The user provides a number of code fragments which define the problem's data structure and some operations on it. For example, one of the code fragments may define when a problem has reached its simplest state. The code fragments are compiled together with the pre-supplied skeleton.

An analogy which may be used to illustrate the concept of an algorithmic skeleton is the C 'qsort' function. The essential element of this comparison is that the Quicksort algorithm is pre-supplied to the programmer. All that is necessary to formulate the sort is a number of user written parameters. One of these parameters is itself a function that describes the comparison to take place.

```
void qsort (void *base, size_t nelem, size_t width, int (* fcmp) (const void *, const void *))
```

The function above defines the Quicksort. It sorts *nelem* entries of size *width* located at *base* and with a ranking determined by a user defined function *fcmp*.

---

1 Programmer (or User) refers to the person writing application programs making use of Algorithmic Skeletons.

The Quicksort function is a very specific solution, whereas skeletons will enable a user to solve a range of different problems. They provide, for example, the general form of a divide and conquer algorithm.

Parallel programs are particularly appropriate for this technique because of their immense level of complexity. The following section discusses some factors that precipitate this complexity.

It should be noted that, as a skeleton hides all the implementation details from the user, it is possible that the code could be placed on a single processor, in which case it will execute sequentially.

1.1 Problems with Parallel Programming

As hardware advances improve the performance, cost, and communication ability of microprocessing devices, it will become possible to build multi-processor computers with greater numbers of processing nodes. As the number of processors in such systems grows, it will become difficult to specify parallel programs efficiently without machine intervention. Therefore, programming techniques will have to adapt in order to relieve the programmer of the more complex issues.

Five complex issues which face the programmer of multi-processor[QUI87] systems are:

- Decomposition - Problem decomposition, or the identification of parallelism in an algorithm, is complicated. Either the user must identify parallelism whilst designing an algorithm, or the system must identify the parallelism. This may be done statically, by analysis, or dynamically at run time.
- Distribution - Distribution of the decomposed processing partitions, which have been identified, over the processors requires a good understanding of the interactions between different components in the system. It must be done statically, at compile time, or dynamically, at run time, by the computer, or coded by the user.
- Sharing - Code and data must be shared by parallel processes or a copy must be made for each process which uses it.

- Synchronization - In order to produce a result, work done by individual processes must be communicated to other processes as they need it. Processes will have to synchronize to facilitate communication.
- Termination - Detecting the termination of a parallel program can be complex. This is because all sub-processes making up the program must be terminated correctly.

It is these complications which the skeleton attempts to solve for the user. Each skeleton presented will be analyzed according to how it deals with these problems. The objective is to give the programmer a "black box" which produces parallel code (see Figure 1).

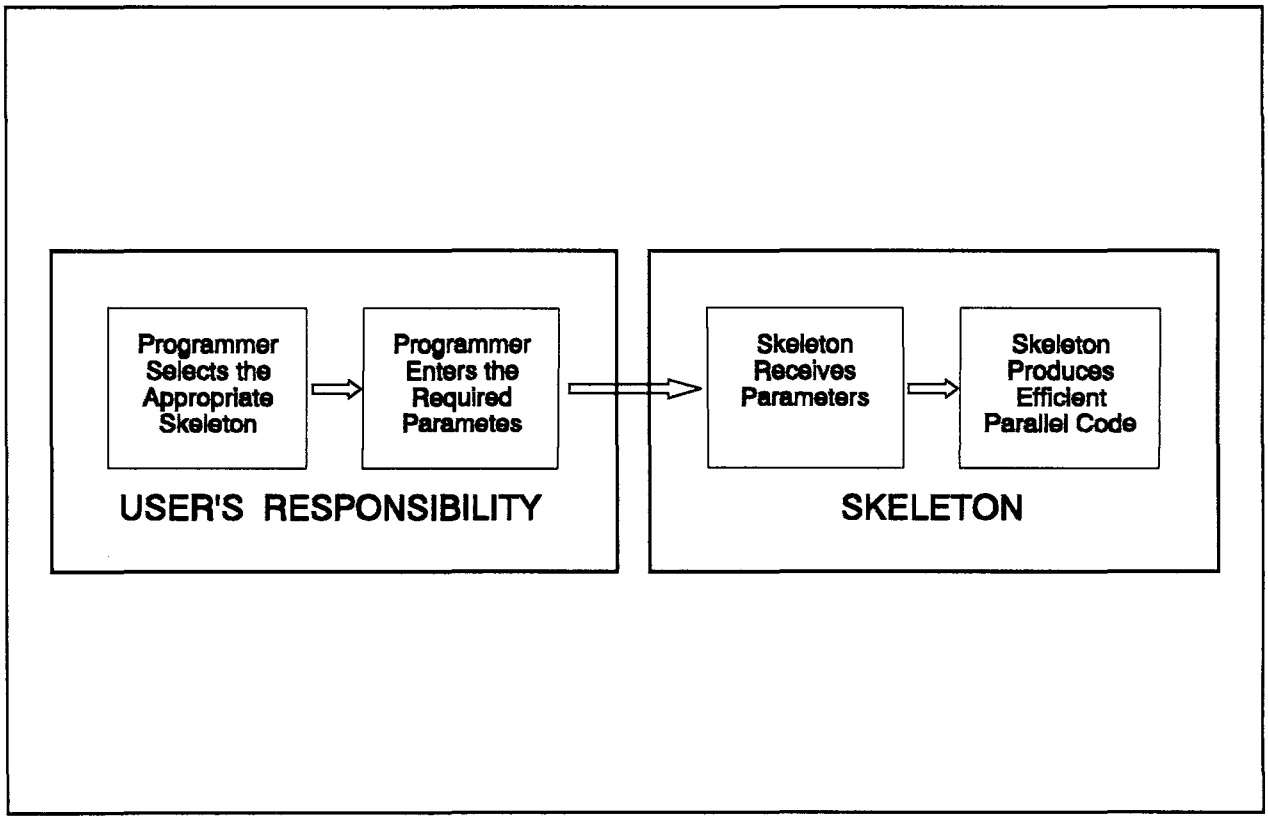

Figure 1 - Schematic representation of the development of a parallel program using an Algorithmic Skeleton.

1.2 Design of a Parallel System

In designing a parallel system, a pressing question is the extent to which parallelism should be reflected in higher level abstractions. Three levels may be distinguished.

Classification of the methods of expressing parallelism:

- Implicit Parallelism - The high level program should show no parallelism, the computer has full responsibility. Highly abstract languages are an example of this category. Functional languages such as Haskell or Data-flow languages fall into this class. Here there is no explicit notion of sequence. However, analysis of the time complexity is difficult, there are unpredictable overheads[KEL89], and performance penalties[FEL91].
- Hybrid - This is an intermediate level. The solutions presented must include explicit parallel constructs. This approach is more amenable to the analysis of time complexity.
- Explicit Parallelism - The user specifies a parallel style closely, for example the Occam language. Overheads are more predictable. The programmer has full responsibility for parallelism[KAC90]. This approach makes portability difficult[FEL91].

Skeletons introduce a new technique for expressing parallelism. They allow the high level abstraction of implicit parallelism, while conserving the efficiency of explicit parallelism. This is because the complicated issues are specified in the skeleton code.

1.3 Towards a Solution

Imperative languages are structured in such a way so as to allow wide categories of problems to be solved. Cole[COL89] described these languages as having "universal constructs". These exist to enable the user to employ the same language constructs to define a range of solutions. This unfortunately gives users the ability to develop algorithms which may be difficult to parallelize. The goal in using algorithmic skeletons

is to admit only those programs for which it can guarantee efficient parallel implementations. The algorithmic skeletons themselves appear non-parallel, hiding complex issues from the user.

An Algorithmic Skeleton may be defined for all familiar algorithmic paradigms. By selecting a particular skeleton the user is unknowingly dealing with the problem of decomposition, as well as selecting an associated pre-defined distribution plan, as these are defined within the skeleton. The user is able to plug user specific code into the skeleton. The notion of higher-order functions are used to allow passing of other functions as inputs to the skeleton (The term higher-order is used here to designate functions which take other functions as arguments or results). A language with a high level of abstraction (with universal constructs) may be used to implement these functions because the problems of decomposition have already been taken care of by the skeleton.

1.4 Implementation Goals

The following desirable properties of an Algorithmic Skeleton system have been identified by means of a literature survey:

- Hardware Independence - The system would be divided into an upper and a lower level. The lower level would map the program onto the hardware, whilst the upper levels would provide the interface to the user. In this way skeletons may be defined for any hardware and any configuration of that hardware.
- Hidden Parallelism - The skeleton would hide all parallel implementation details from the user. The literature suggests that this is not currently possible[PRI88].
- User Interface - A clear and efficient user interface should exist for ease of use.
- Independence - All skeletons should be completely independent of one another. This allows skeletons to be added and removed as the concept is developed[COL89].
- Exclusion - The skeleton should not allow applications for which it is not suitable.
- Performance - There should be performance benefits related to the number of processors in the skeleton system.

1.5 Algorithmic Skeletons Versus Parallel Libraries

The skeleton approach differs from the concept of reusable libraries in that it is rare to find genuinely reusable libraries for MIMD parallel computers[FOS]. They tend to be very specific and help with details rather than give an overall solution. Higher-order functions are not well supported by parallel libraries. In contrast, most skeleton inputs are of this form, which allows for a much wider application. In order to make use of higher-order functions in the development of a skeleton, functional languages, which are inefficient, have usually been employed. This is because higher-order functions are not as intuitively supported in imperative languages.

1.6 The Approach of this Thesis

This project differs from the typical approach to skeletons in that the imperative language 'C' has been employed, in the hope of achieving an efficiency not found when using a functional language. Higher order functions are simulated, using the 'C' text preprocessor to insert the user's functions into the skeleton.

Two skeletons are presented in this report, based on the notions of "divide and conquer" and "task queues"[COL89], [KEL89], [HOR83]. Skeletons under investigation by other researchers[FEL91] include multi-pipelines and geometric decomposition.

1.7 Thesis Outline

This chapter has presented the notion of an algorithmic skeleton as an abstraction designed to aid in the development of parallel programs. The fundamental problems areas in parallel programming have been introduced to highlight the need for a programming aid of this type. The goal of the approach will be to hide some of these problems from the user. This thesis will discuss the development of the skeletons in

terms of the issues they hide from the user and how this is achieved. It will also evaluate the paradigm in the context of an imperative (rather than functional) implementation.

Chapter 2 discusses related research.

Chapter 3 details the hardware and software that was used in the development of this pilot study.

Chapter 4 presents the complete implementation of a Divide and Conquer Skeleton, and discusses how each of the fundamental problem areas have been addressed. This chapter presents four example applications.

Chapter 5 details the implementation of a Task Queue Skeleton. This chapter discusses how the problems of parallel programming are removed from the user, and gives examples for the use of the skeleton.

Chapter 6 reviews an idea by Foster [FOS90a] in which he attempts to create standard skeleton interfaces to allow a problem to be solved with more than one skeleton. In his work he introduces the idea that skeletons may combine features of one skeleton with features of another through composition. The chapter also describes the extent to which the work of this thesis fits his paradigm.

Chapter 7 is an evaluation of the skeleton approach. It shows that, as implemented here, most examples tested scale well with the approach, but that degradation of performance occurs in one test example. The discussion shows how a programmer with some appreciation for the underlying architecture can optimize his results. A performance analysis of the two classes of skeleton implemented is given.

Chapter 8 gives the conclusions reached in this thesis.

Appendix A presents a brief overview of the Helios CDL notation. The implementation was done on transputers running in the Helios operating environment. Some of the sample code requires a knowledge of the Helios CDL notation.

The full source code for this thesis is not included in this document, and is available from Rhodes University Computer Science Department.

Chapter 2

2. Related Research

The research into skeleton-type programming aids is very broad. This chapter discusses three concepts related to algorithmic skeletons, each of which addresses the issues in a different manner from the implementation of this thesis. Other areas of research more closely related to this implementation are referenced in the thesis where necessary.

Concurrency: Simple Concepts and Powerful Tools [FOS90b]

Foster et al. describe an approach which differs from the algorithmic skeletons methodology described in chapter 1 in that the code is not actually written for the user. Rather they are introduced to specific programming concepts and shown the techniques for parallel programming which are derived from these concepts.

In order to simplify parallel programming and allow reuse of code the authors suggest methods to allow the concept of stepwise refinement to be successfully applied to parallel program design. Stepwise refinement implies that a program is successively refined from its initial specification, allowing separate aspects to be dealt with as distinct steps. In this way decisions can be delayed. In terms of parallel programming these decisions would pertain to issues such as decomposition and mapping. This results in increased independence and hence allows for modification and ease of porting. This is only possible if preceding steps don't commit the program to a specific architectural stance, in terms of communication synchronisation and concurrent execution.

The analysis of communication and synchronization identifies six programming techniques or the development of six skeletons programming methods:

- Producer/Consumer Protocol - This protocol allows for unbounded communication between a single producer and many consumers.
- Incomplete Message Protocol - This protocol allows for two way unbounded communication.
- Bounded-buffer Protocol - This protocol organises communication so as to bound the number of unreceived messages.

The above are all stream based inter-process communication protocols.

- Difference List - The difference list allows for list constructed by many producers.
- Short Circuit - Short Circuit allows for the detection of termination of the program components.
- Monitor - This concept permits concurrent atomic access, to a shared data structure.

These programming techniques are made possible by four ideas. They are the concepts of monotone variables, concurrent interleaving, non-deterministic choices and separation of sequential code. Through these concepts the authors believe that architecturally general designs can be achieved.

Monotonicity - A monotone variable can be assigned a value once only. This variable can be used for communication (blocking) and synchronisation. They can be easily implemented and allow strict reasoning regarding the program.

Interleaving - Code segments are guaranteed to execute, but the order in which they execute is not constrained. As it is not important when code executes the authors suggest that decisions on decomposition, mapping and granularity are isolated.

Non-deterministic Choice - There must be choice between various actions. Monotonicity allows for reasoning regarding choice made.

Separation of sequential code - Black boxes of sequential code are allowed to enable use of destructive operations on variables and efficient use of memory. This is made available by simple interfaces to conventional languages.

This approach meets the goals of algorithmic skeletons in so far as it provides simplicity of coding, reuse of past work and portability of code.

They conclude that "languages do not solve problems, they merely provide a means of describing solutions. Design and implementation of problem solutions are primary a programming task and must be supported by an appropriate methodology. "[FOS90b]

Towards a skeleton based parallel programming Environment[FEL91]

Feldcamp et al. describe an approach to skeletons which allows for application tuning. They identify the two most important features of a skeleton as that it should provide an underlying structure that is hidden from the user, and that this structure should be an incomplete module requiring parameters. Although it is important to provide this hidden structure, they suggest that important machine dependent features cannot be ignored because of their impact on performance. These are features such as granularity and topology. By providing a simple interface, they hope to create a system in which the user can concentrate on the computational task, rather than on control and coordination of parallelism.

An important feature of their approach is that they provide models which can be used to identify causes of inefficiency. In some cases improvements are achieved via adjustments to user input parameters. Their parameters for tuning a skeleton are:

- Number of task packets per communication.
- Maximum number of tasks assigned to a packet.
- The number of processors and their interconnection (the default arrangement of their processors is that of a tree).

They explain that there are a relatively small number parallel programming paradigms, and describe implementations for the Divide and Conquer Skeleton and the Farm of Worker Skeleton.

They describe the important features of a skeleton as:

- Simple user interface.
- Performance monitoring tools. These show how a program is performing.
- A simple way to integrate user code.
- Special-purpose visualization of performance results to allow for easy understanding.

Their results suggest that altering the number of tasks on various nodes can improve performance, and that an optimum granularity exists for tasks. Their model fails when problems become communication bound.

Reusing and Interconnecting Software Components [GOG86]

This approach suggests the construction of a database to store segments of code for reuse. Through this Goguen [GOG86] hopes to make programming easier, more reliable and more cost effective. The system would allow for programs to be built using code segments from the data base. High-level paradigms for program design include:

- Top down - Program design may occur in a top down fashion, starting with the application requirements and working towards the data base library modules.
- Bottom up - Program design starts from the selected library modules and develops towards the application requirements.
- Transformations - A program may be formed from transformation of a prototype.

This differs from the concept of algorithmic skeletons in that algorithmic skeletons propose the pre-supplying of a limited number of algorithmic paradigms as general purpose frames, and not masses of specific code modules.

An analogy to this approach is to view software like Lego[2]. A piece of Lego may be connected to another in a number of ways, depending on a matching between it and the other piece. Many different objects can be build in this way, by reusing the components.

There are a number of issues to be dealt with in the creation of such a data base, in order to build a consistent environment that is easy to understand and use:

- Selection of entities for storage.
- Techniques for program composition.
- Documentation and specification techniques.
- Identification of software components.
- Relating programs.
- Integration of facilities.
- Presenting information to the users.
- Viability tests.

The development of a data base of software components for reuse requires clear specification for possible interconnection of components. Methods of interconnection and formal validation are required. Goguen suggests the adoption of the following design considerations:

- Views - Views describe the semantically correct interconnection of software components. Systematic meaning is given to software components by the explicit attachment of theories to the code.
- Generic entities - Reusability can be maximized by the use of generic programming methods.
- Composition - The paper distinguishes between horizontal and vertical composition. Vertical composition deals with the development of a hierarchy of abstraction. Horizontal composition refers to modularization at a given level.

---

2 Lego is a child's toy which consists of plastic bricks of different shapes.

- Interconnection - The use of a library interconnection language (such as LL) to construct large programs form existing entities.
- Formality - Formal methods in documentation of software components and validation will be used.
- Abstraction - Abstract data types are used to facilitate the understanding of programs.

The Approach of this Thesis

This chapter has presented a number of ways in which the reuse of a programming framework may be applied to ease, and to speed up program development. The approach used in this thesis is the construction of a number of skeletons, each based on frequently used algorithmic paradigms. In this way users do not have to redevelop common portions of code, but merely provide code fragments specific to the current problem.

The implementation of this thesis differs from the work of Foster et al. and Goguen in that an algorithmic paradigm is implemented for the user. In the approach of Foster et al., aid is given to the user for the construction of an algorithm. Goguen requires the user to develop his own algorithm, while supplying specific code portions for that algorithm.

Feldcamp et al. describe methods of tuning a skeleton implementation for better performance. This is an issue not fully addressed in the implementation of this thesis. In the evaluation (chapter 7) tuning is suggested as a solution to specific problems.

Chapter 3

3. Hardware and Software

In order to understand the implementations presented in this thesis it is necessary to discuss the hardware and software used to create the skeletons themselves. The implementations were hosted on a network of transputers running under the Helios [HEL91] operating system. The transputer architecture is discussed in this chapter, together with a brief description of the operating system and languages used.

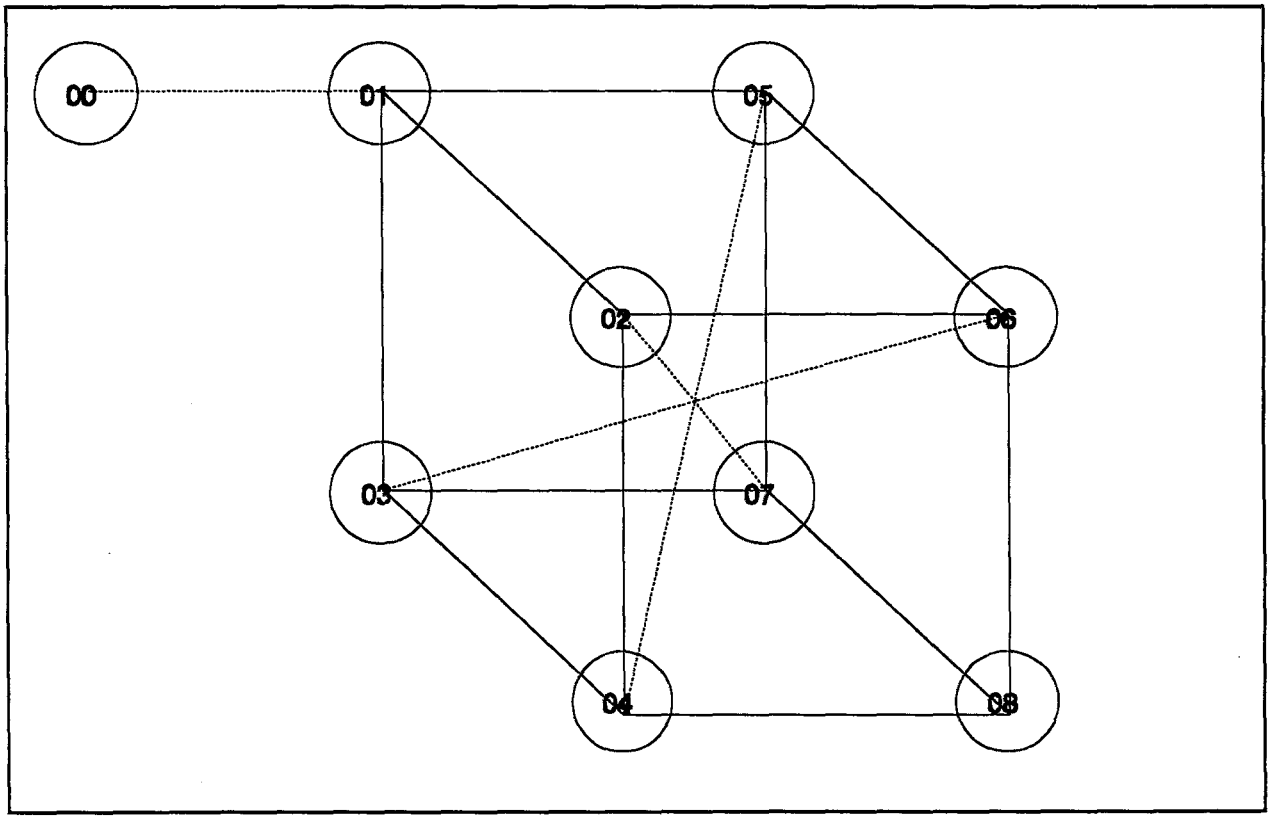

Figure 2 - Hardware Configuration.

3.1 The Transputer(T800) - a brief description [TRA89]

The transputer is a single chip, risc computer. It executes sequentially. It has 4k on-board memory, but can address 8 gigabytes of external memory. There are 4 input/output communication channels (links). Communication over these links is serial and operates at 10 or 20 megabits per second (mbps). The links are autonomous. When communication is to take place between two transputers, the communicating process is descheduled and a DMA controller handles the data transfer.

Nine transputers, located in a Maxi-Cluster[3] computer, were available for this research. The Maxi-Cluster allows for all nine transputers to be configured, by means of soft-switching, in an arbitrary topology. In a network of transputers, each transputer operates asynchronously. The Maxi-Cluster is connected to an IBM PC compatible microcomputer, for I/O purposes.

3.2 CDL - Placement and Communication

A Unix-like operating system called Helios runs on the transputers. Helios provides a process description language called CDL (Component Distribution Language) to allow for distribution of code over the transputers as well as the setting up of communication channels.

CDL allows the user to specify a task-force of arbitrary logical topology along with communication and other requirements. In general, this task force is independent of the size and physical topology of the available resources, as CDL handles the mapping and routing of messages. CDL also allows explicit hand-coded placements, which this thesis uses. These enable a programmer to specify the configuration of the tasks, and to place them in the most appropriate manner. Code can be assigned to any processor in the network. The physical channels can also be directed using the CDL script. In this way a virtual tree is created for the Divide and Conquer Skeleton and a farm of workers for the Task Queue Skeleton, while ensuring a nice fit between the virtual implementation and the physical topology.

---

[3] The Maxi-Cluster is classified as a MIMD (Multiple Instruction, Multiple Data Stream) computer according to Flynns Taxonomy[QUI87].

Helios uses a resource map for information regarding the Maxi-Cluster's configuration. A standard 3-D cube configuration, which allows for the maximum interconnectivity of the available transputers, is shown in Figure 2. Additional connections are made between diagonally adjacent processors. This configuration is used as a standard topology for all skeletons tests developed for this thesis. Processor 00 is connected to the host PC and has only two links available to the user (see section 3.1). Processor 01 is therefore used as the root processor in this thesis.

The CDL level offers the benefit that the distribution algorithm for a skeleton, implemented in CDL script, can be decoupled from the skeleton code. If necessary, a new algorithm can then be substituted, with the change being transparent to the skeleton code (see Figure 3). Naturally any inbuilt topological dependencies of a skeleton should be reflected in the CDL level. If the skeleton expects to operate on a tree structure, the underlying configuration must reflect that structure.

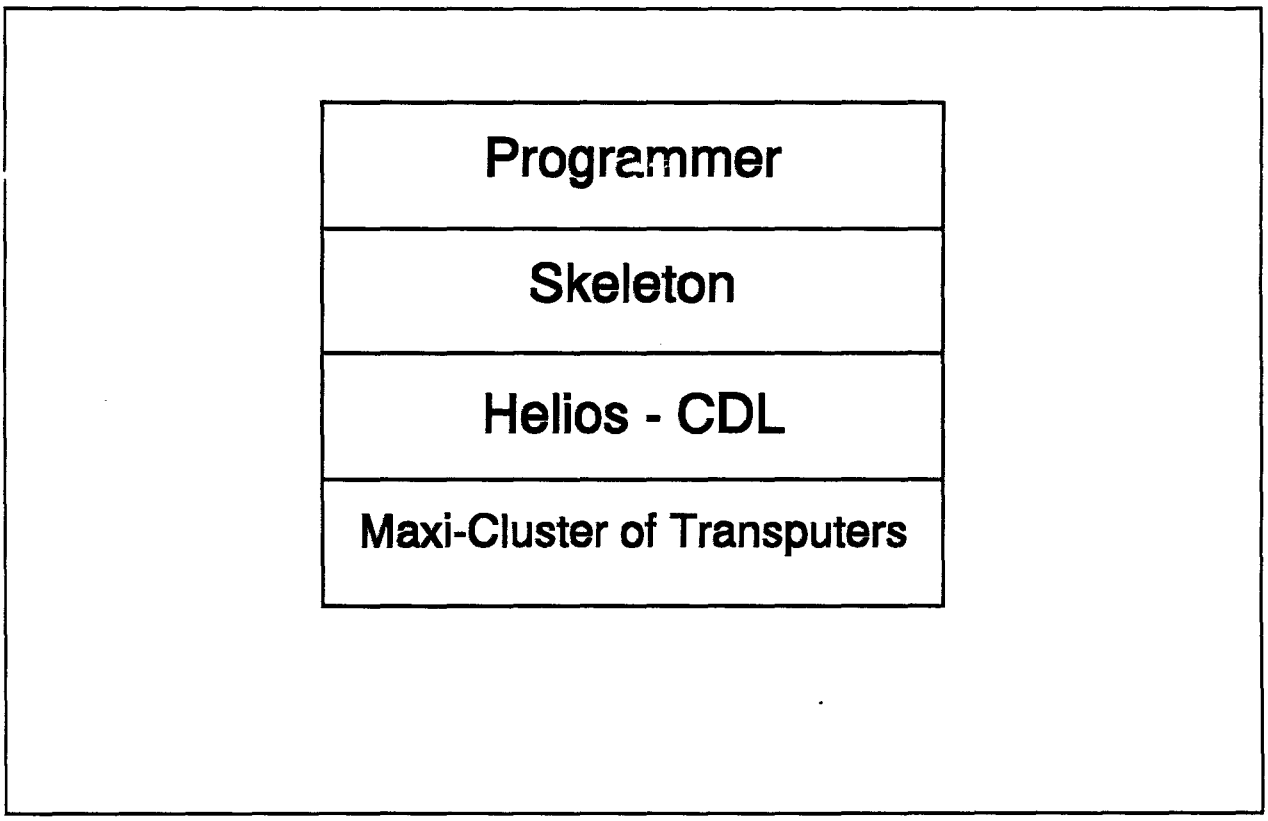

Figure 3 - Levels of Abstraction.

3.3 Chapter Overview

This chapter has given a brief description of the hardware and configuration software used in this work. The project was implemented on a Maxi-Cluster of transputers configured using soft switching. An Unix-like operating system, Helios, runs on the transputers. CDL is provided by Helios for specifying the creation of a user task force. This task-force is placed over the actual hardware of the Maxi-Cluster and may represent any possible configuration. CDL can route messages if a direct mapping is not possible. In the implementations of this thesis, a direct mapping has been specified to avoid the overhead of routing messages.

Chapter 4

4. Divide and Conquer Skeleton

The divide and conquer scheme is a useful technique for solving complex problems. Under this scheme the solution to some problem may be found by dividing it into its component parts and solving each sub-part in the same manner. When a sub-problem is indivisible, it is in its simplest form and may be solved directly. These simple solutions may be combined to form a single answer. This algorithm naturally embodies the concept of parallelism as each sub-problem may be solved simultaneously and independently.

Examples of problems which may be solved using the divide and conquer technique are Quicksorts, integration problems, and matrix multipliers.

The algorithm in Figure 4 finds the maximum and minimum of a list of integers, using the divide and conquer technique. This example will be used to show the development of the Divide and Conquer Skeleton.

4.1 Application Specific Code

In order to develop a skeleton to describe the divide and conquer paradigm the essential elements of the scheme must be distinguished. The following distinct parts can be identified [KEL89], [RAB90], [FEL91], [HOR78]:

- Trivial - The Trivial Case is a Boolean expression to determine whether a problem is in its simplest form. In Figure 4 the problem is in its simplest form when the list is of length 1.
- SimplySolve - A Simplest case routine, which directly solves the trivial case.
- Decompose - A routine to divide a complex problem into simpler components.
- CombineSolutions - A routine to develop an answer from a set of simpler case answers.

```
/*    A procedure to find the maximum and minimum of a list
      using the Divide and Conquer technique.
 */

Proc MaxAndMin (List)

if length (List) = 1 then
      return (a, a) where the List = {a}
else
 let
      List1 and List2 = partition List into two
      (Max1, Min1) = MaxAndMin (List1)
      (Max2, Min2) = MaxAndMin (List2)
 in
      return (max (Max1, Max2), min (Min1,Min2))
```

Figure 4 - The MaxMin Problem.

These are the components of the divide and conquer which change between each application of the divide and conquer paradigm. Therefore according to the philosophy of algorithmic skeletons it is these portions of the code which may be extracted as user input. The following routines are specific instances which solve the MaxMin problem.

- Trivial Case: if length (Problem) = 1.
- SimplySolve: return (a, a) where the list = {a}.
- CombineSolutions: return (max (Max1, Max2), min (Min1, Min2)).
- Decompose: divide Problem into Prob1 and Prob2.

The users are constrained to mould their solutions around these four code fragments, the types of which are illustrated in Figure 5. Each code fragment will receive some problem data of a specified type. The user is expected to perform some manipulation on this data and return a solution of a required type. In the above example[4] the code fragment Decompose receives a problem list and is expected to divide it into two sub-lists. It must return these two lists as its solution.

4 Example application programs are discussed under section 4.9 Applications.

Type definitions for the user code fragments:

- Trivial - Trivial is a function which takes a problem of type α and returns a Boolean value.
- SimplySolve - SimplySolve is a function which takes a problem of type α and returns a solution of type ß.
- Decompose - is a function which takes a problem of type α and returns a list of problems of type [α].
- CombineSolutions - is a function which takes a list of sub-solutions of type [ß] and returns a solution of type ß.

```
DCSkeleton : is a function which takes four arguments

  Trivial   : α -> Bool
  Simplest  : α -> ß
  Decompose : α -> [α]
  Combine   : [ß] -> ß

and it returns a new function
  DivideAndConquer : α -> ß

In all these signatures, α stands for the type of the
problem, ß for the type of the solution.
```

Figure 5 - The interface specification of the skeleton inputs.

4.2 Common Divide and Conquer Code

The portion of the example algorithm which is common to all divide and conquer algorithms can be extracted. It remains the same, requiring the user code fragments to form an application. The high-level code which implements the divide and conquer mechanism is shown in Figure 6.

```
f(x) = DivideAndConquer (Trivial, SimplySolve, Decompose,
                         CombineSolutions, Problem)

    if Trivial (Problem)
    return (Simplest (Problem))
 else
    let SubProblems = Decompose (Problem)
        SubAnswers = map*(DivideAndConquer, SubProblems)
    Result      = Combine (SubAnswers)
 in
    return (Result)


* Apply function DivideAndConquer to all SubProblems
```

Figure 6 - Divide and Conquer Skeleton Code.

It is this framework of the divide and conquer algorithm that the skeleton designer must implement efficiently in parallel. A formal specification of the Divide and Conquer function is shown in Figure 7. The Divide and Conquer function takes the functions Trivial, SimplySolve, Decompose, Combine, a problem α and returns a solution ß.

```
DivideAndConquer :   (α → Bool)     /* Trivial      */
                     (α → ß)        /* SimplySolve  */
                     (α → [α])      /* Decompose    */
                     ([ß] → ß)      /* Combine      */
                     α              /* Problem of type α  */
                   → ß              /* Solution of type ß */
```

Figure 7 - Type specification for the inputs and output of the Divide and Conquer Skeleton.

4.3 Solution Development

The solution to a divide and conquer algorithm develops as a single problem enters through the root process. The problem is split into its sub-components. Each component is further split into yet simpler components until the trivial case is reached. This process creates a tree structure which may be understood by examining Figure 8.

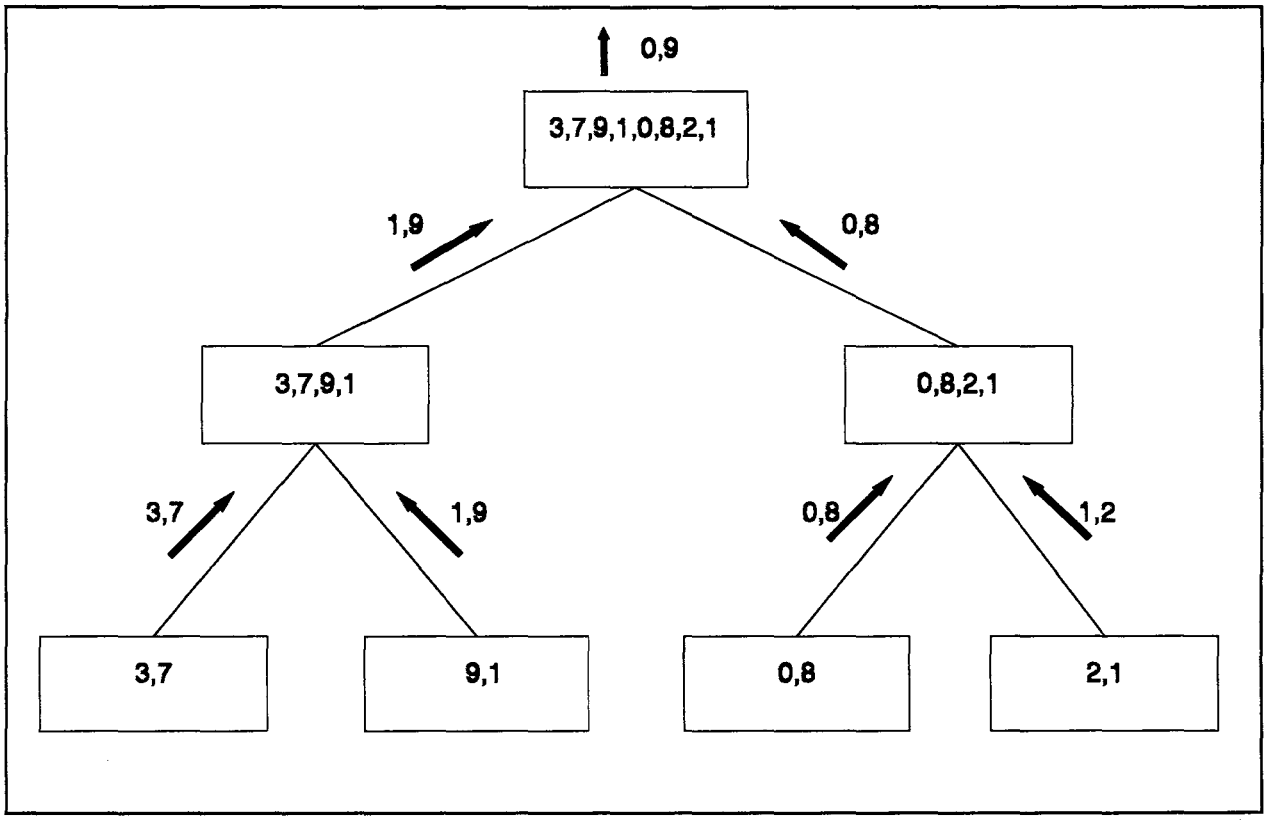

Figure 8 - Finding the maximum and minimum of a list using the Algorithm of Figure 4, given the input list shown in the root process.

The larger the original problem the larger the solution tree will be (i.e. The amount of potential parallelism depends on the problem size). In a multi-processor environment each of the successive problems may be solved independently on a different processor with the solutions passed back to the parent processor. Placing each new sub-problem as an independent task results in a need for communication between a parent task and its children. It also means that a parent must wait for solutions

from its children. These are issues which must be addressed by the skeleton. It is the task of the skeleton to ensure that an efficient mapping of this dynamically evolving tree occurs.

The following four sections will discuss how the skeleton tackles some of these difficult questions.

4.4 Distribution

The central issue of this section is the decision on how distribution should be handled. Various distribution algorithms were considered in terms of a number of criteria (The first two points are also presented in Boillat et al. [BOI87]):

- Efficient load balancing - When dealing with a large number of parallel tasks it is important to handle load balancing in an efficient manner. This is discussed in section 4.4.2.
- Limited interconnection - Transputers have four communication channels, making it difficult to connect all processors in a network to each other. In order to limit routing of messages it is important to make maximum use of the available connections (section 4.4.1).
- Limited number of processes - It was important to make optimum use of the nine transputers that were available for this research (section 4.4.2).
- Easily scalable algorithm - As more processors become available they should be easily added to the skeleton system (section 4.4.3).
- Simplicity of implementation - A simple algorithm was preferred for this pilot study into the use of skeletons.

These criteria are taken into account in the implementation of the distribution algorithm and are discussed in more detail below.

The ideal distribution is a k-ary tree[COL89], as shown in Figure 9. Mapping this directly onto a network of processors would be very complex in terms of the restrictions listed above, therefore a modest version of this algorithm was devised.

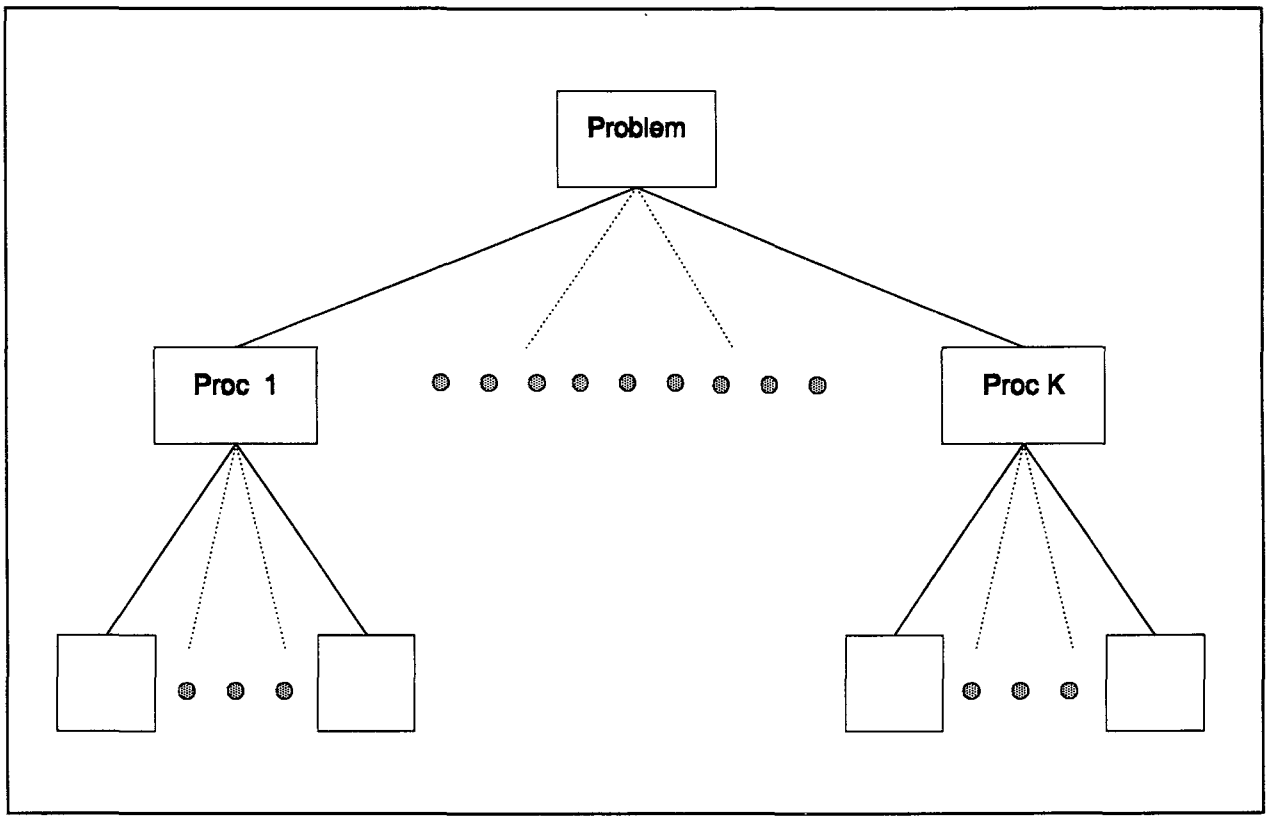

Figure 9 - Divide and Conquer Problem of degree *K*.

4.4.1 Limited Interconnection

Limited by 4 transputer links, a tree with a maximum degree[5] of three may be implemented (A node requires a link for communication with the parent leaving 3 links for communication with children). A binary tree is the most common form of tree[HOR76]. It is the base case because it is possible to convert any tree to a binary tree. A tree of degree two was therefore implemented for distribution of the Divide and Conquer Skeleton.

5 For the purposes of this thesis, degree refers to the number of sub-problems into which a problem may split.

4.4.2 Load Balancing with a limited number of Transputers

A binary tree implies that at each level of the tree the problem may be halved (see Figure 10). Each time the problem is decomposed it is placed on a new processor.

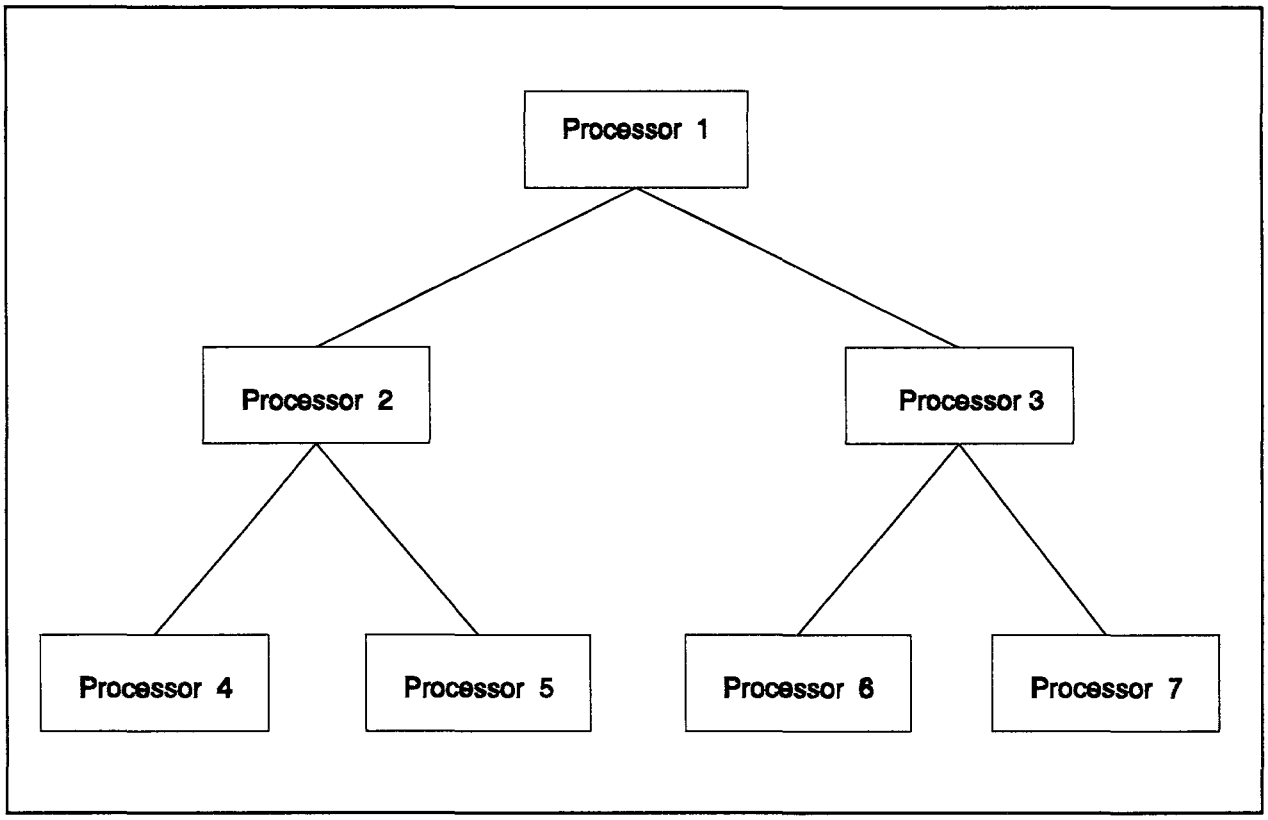

Figure 10 - Divide and Conquer problem of degree 2.

As the problem filters through the tree the parent nodes will be inactive for most of the time, while waiting for the results from their child tasks. An algorithm which maintains an appropriate load on the parent transputers is needed. When the parent task enters a waiting state its processor can be rescheduled so as to operate on one child (For future clarity P-Child will be used to refer to this child). In this way all transputers are kept active. Figure 11 illustrates how the tree of Figure 10 is implemented on only 4 processors. Child tasks 2 and 4 now execute on processor 1. Child task 6 executes on processor 3.

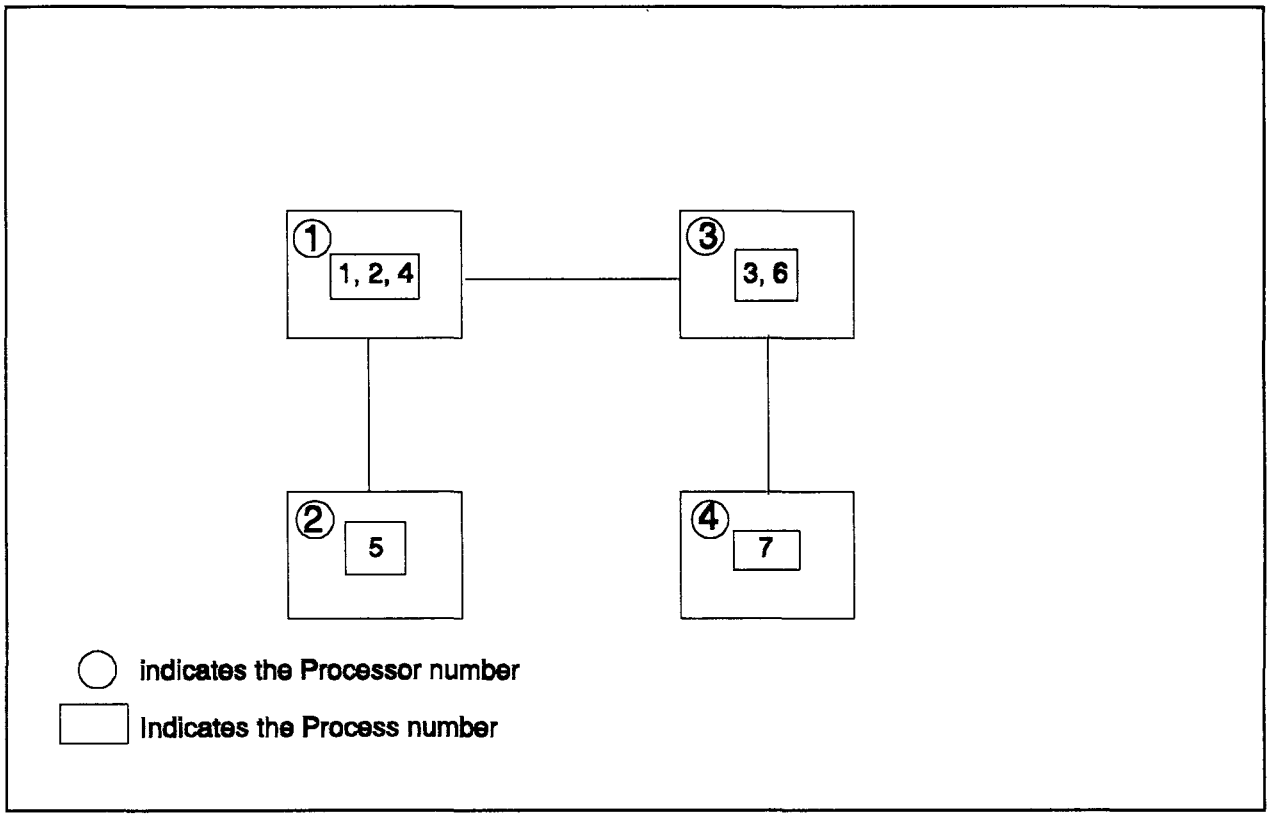

Figure 11 - As soon as a parent becomes inactive one of its children may execute on its processor.

In order to create a more intuitive implementation, P-Child is sent to a pseudo-processor, which is a process on the parent transputer (see Figure 12). Leaf nodes are reached when the tree is at its maximum depth and no more transputers exist (Once a problem reaches a leaf processor, further sub-solutions are obtained by means of recursion).

The notion of pseudo-processors is for the purposes of this thesis only. It is intended to emphasize the tree structure of the algorithm and to provide an increased clarity to the reader, even through it may not be more efficient. Task-switching is hardware controlled on a transputer and is very efficient. Communication to processes on the same processor is also efficient. An additional advantage of this extension is that input and output channels between parents and children can be standardized, because all child process appear the same to the skeleton. It is only necessary to specify that a particular child process must reside on the same transputer as its parent.

In the event of user inputs with a degree other than two, a number of the children are sent to each channel so a transputer now handles a queue of data items and not just one. This feature allows irregular trees to be load balanced, as the portion of data passed to any child may be varied depending on the processing requirements. The default portion is set at a half, but may be altered by the programmer. This violates a fundamental goal of algorithmic skeletons because the user now needs explicit knowledge regarding parallelism.

Three inputs are required by the skeleton to enable it to make decisions regarding the flow of problem data (see Figure 14). The second two are part of the skeleton implementation, and are not user supplied.

- The skeleton must receive the degree of the user application as input.
- A specific transputer must be informed as to how many processes it is to handle. The root process will only receive one initial problem to solve. Parent processes send one problem to each of n children, where n is the degree of the process tree. Each child process will in turn generate n sub-problems. At each stage, at least one child process will reside on the same transputer as its parent.
- A process must know whether it resides on a leaf transputer (any further sub-problems will have to be evaluated locally).

4.4.3 Scalable

This distribution does not make maximum use of all transputer links. Figure 12 shows that all links of processor 1 have been used, whilst processors 5, 6, 7 and 8 have three unused links each. Further processors may be added, but, as the tree grows, increasing use will have to be made of Helios message routing. A more experienced programmer can alter the amount of data sent to each processor to reduce any resulting bottlenecks.

The CDL code is used to explicitly place processes on the correct transputers and to set up the links between them. Processes are placed according to Figure 12. The distribution has been implemented in the form of a removable module to enable testing of the usefulness of Algorithmic Skeletons, and also to allow further development or a complete change of the algorithm.

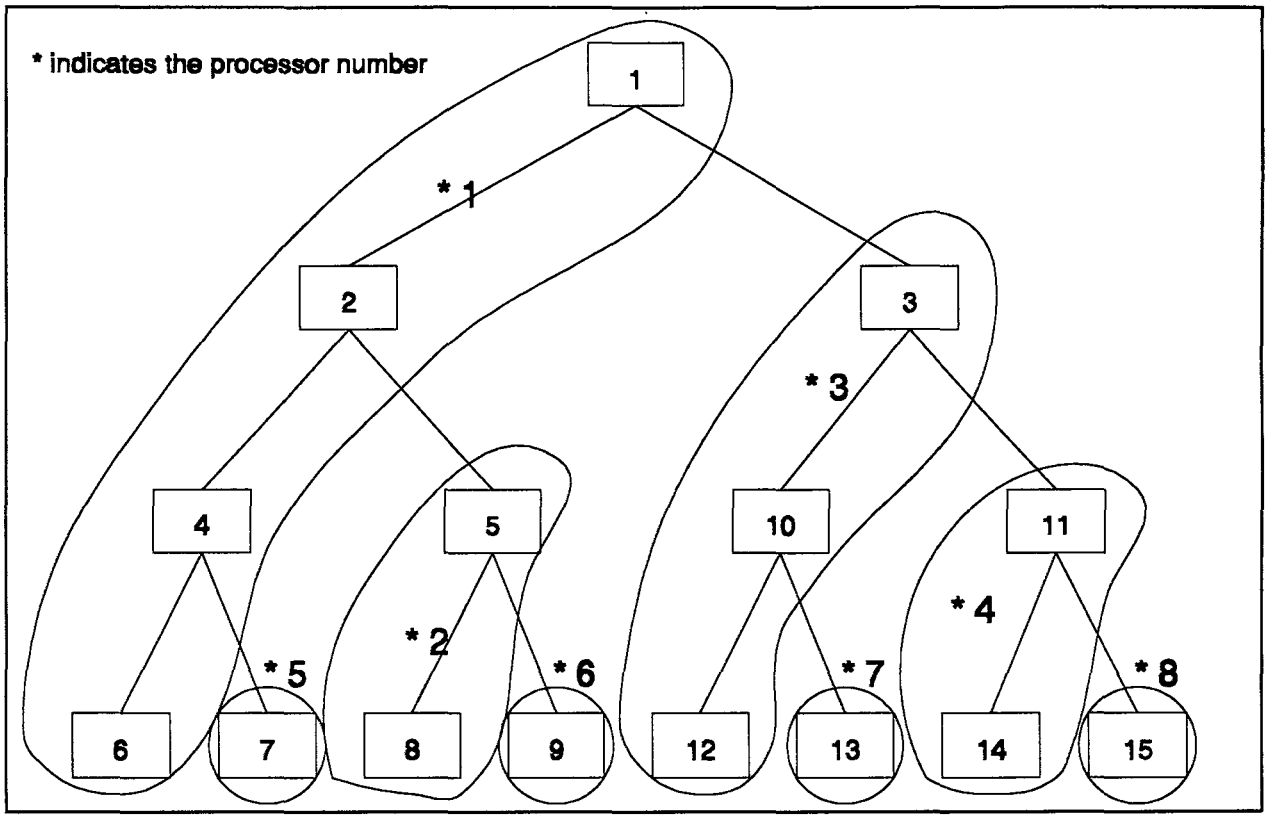

Figure 12 - Process Location.

4.5 Communication

Another concern of algorithmic skeletons is the efficient and simplified transfer of user data within the skeleton, as well as the transfer of data to and from the user. The issue of communication is complicated by the fact that it may take place between a user data structure on one transputer and the user data structure on another transputer. When programming using a MIMD[6] architecture it is preferable to store data in contiguous segments of memory, as block transfers of data are more efficient. The skeleton provides communication procedures to deal with the block transfer of a data structure. Should a user's data not occur in this form, it would be his responsibility to pack the data into a contiguous message, and then use the skeleton block communication.

---

6 Multiple Instruction Stream, Multiple Data Stream [QUI87].

Implementation

CDL is used to establish channels between processes, using the Unix channel conventions. The Divide and Conquer Skeleton uses two way channels between processes. Each process is placed on its transputer and the links are opened to complete the tree structure. The first process (input/output process) is responsible for reading in the initial problem and writing out the solution. It has a stdin, stdout, stderr connected to the console (see Figure 13). The first processing node reads the problem on stdin and writes the solution to stdout. It writes the sub-problem to be processed to channel 4, and reads the solution from channel 5. All processing nodes in the tree have their stdin/stdout connected to their parents output/input channels. A processing node writes to and reads from channels 4 and 5 (left sub-tree) and 6 and 7 (right sub-tree). Table I shows a textual representation, and Figure 14 is a graphical illustration, of these channels .

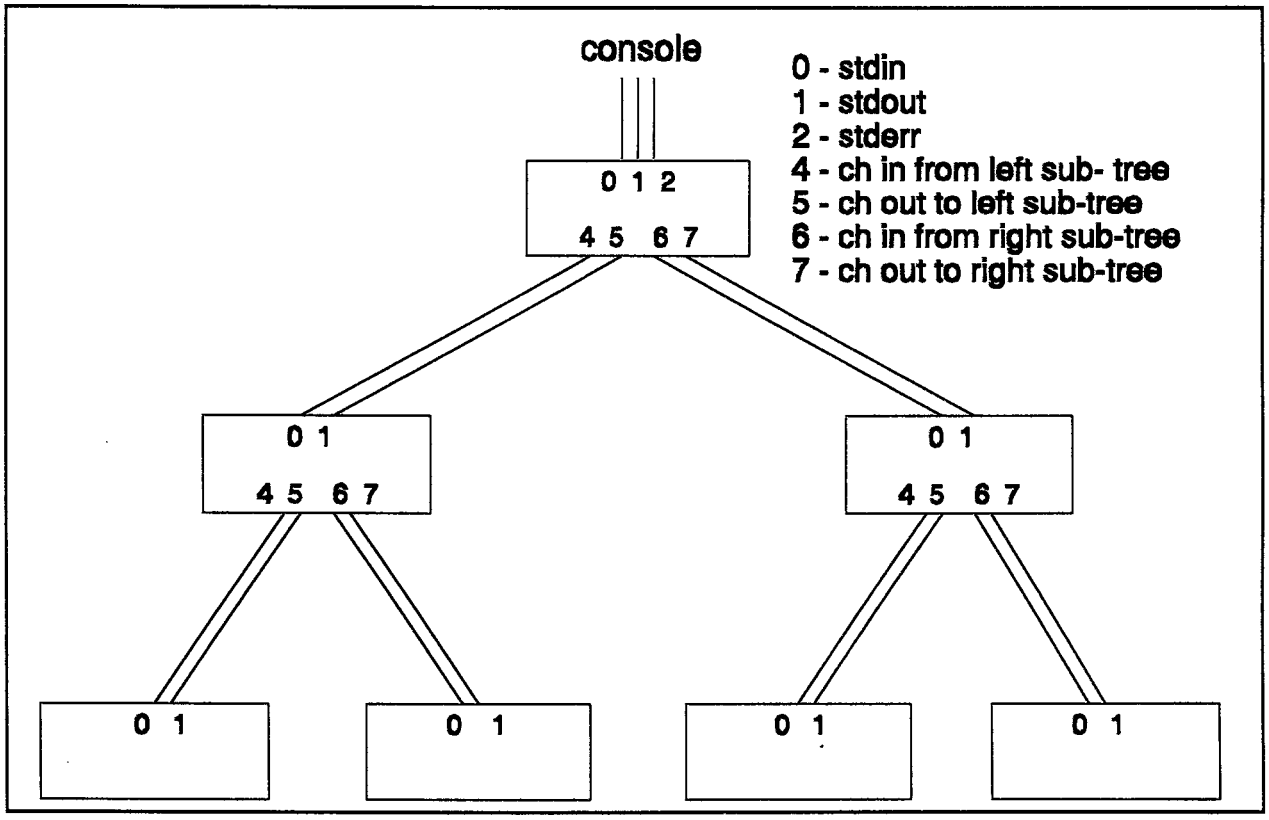

Figure 13 - Illustration of Process Channels.

Table I - Skeleton Communication Channels.

File Descriptor	skc	sk_1	sk_2
0	console	input from skc	input from sk_1
1	console	output to skc	output to sk_1
2	console	console	console
3	unused	unused	unused
4	input from sk_1	input from sk_2	input from sk_4
5	output to sk_1	output to sk_2	output to sk_4
6	input from sk_3	input from sk_5	
7	output to sk_3	output to sk_5	

File Descriptor	sk_3	sk_4
0	input from sk_1	input from sk_2
1	output to sk_1	output to sk_2
2	console	console
3	unused	unused
4	input from sk_6	
5	output to sk_6	
6	input from sk_7	
7	output to sk_7	

sk_5, sk_6 and sk_7 are similar to sk_4

The CDL notation is reasonably powerful. The simplified code in Figure 14 implements the structure discussed.

4.6 Synchronization

Synchronization of the various tasks is closely linked to the transfer of data. A task must wait for data to operate on and must write out sub-problems if need be. It must wait for the return of results. Synchronization is thus coupled with data transfer.

```
/* ------------------------------------------------------

        Cdl code defining the Process Tree:

                        Input Processor
                              |
                          Process 1
                         /         \
                    Process 2       Process 3
                   /         \     /         \
          Process 4   Process 5   Process 6   Process 7
          / \           / \         / \         / \
     Leaf 1 Leaf 2 Leaf 3 Leaf 4 Leaf 5 Leaf 6 Leaf 7 Leaf 8

 * ------------------------------------------------------ */

     CDL CODE

     skc <>
       sk1 8 1 0
         (<> sk2 8 4 0 (<> sk 8 16 1, <> sk 8 16 1),
          <> sk3 8 4 0 (<> sk 8 16 1, <> sk 8 16 1)
         )
```

Figure 14 - Simplified CDL notation for Skeleton Distribution.

As CDL adheres to the Unix conventions; its reads are non-blocking, and its writes block. It is therefore sometimes necessary to implement a "wait for data" procedure to cause a process to suspend until data is available.

4.7 Termination

A process will wait until all expected problems are received, as discussed these will number n^k, where n is the degree of the problem, and k is the processes's level in the process tree. Once all expected problems are received the process will terminate. A NULL problem is sent out if a problem reaches its trivial state before it reaches the bottom of the process tree. This will cause all processes lower in the tree to terminate.

4.8 The User interface

The above specifications are defined at a very high level. Before a truly useful skeleton can be implemented in true parallel style, a clear and efficient user interface must be defined. This interface must allow for the efficient passing of data and simple specification of user routines.

Before a user interface can be developed, the type of input a user may wish to give the skeleton has to be clearly understood. Unfortunately this interface is inevitably a compromise, to a greater or lesser extent, between the smooth workings of the skeleton and the needs of the user.

In the implementation described above, the user is provided with five skeleton procedures, with their pre and post conditions specified. In this way it is clear what input is expected and what results are required for a procedure.

4.9 Applications

Four example applications are discussed below to illustrate the use of the Divide and Conquer Skeleton. An important test is to determine whether programs execute between some expected bounds (worst case to best case).

Data transfer[HOR83] between different processors adds a significant time component in a parallel applications. Therefore, the computational complexity of the algorithms must take into account their time complexity, as well as their data movement complexity. Data movement complexity is based on the amount of data that needs to be moved between the local memories of the various transputers.

4.9.1 Integration

In order to integrate a non-negative function, the area under its graph can be approximated by a number of rectangles. As the number of rectangles grows, the accuracy of the solution improves. Given a

```
Area

The area under the graph of a non-negative
continuous function f over an interval [a, b]
is the limit of the sums of the area of
inscribed rectangles of equal base length
as their number n increases without bound.
```

Figure 15 - A definition for the integration example.

function, the integral can be successively broken into a sum of smaller integrals using the divide and conquer algorithm. Each integral can be solved or divided again independently of the others. These independent solutions are then added to give the total integral. A skeleton was developed[7] to find the area under a circle (see Figure 15).

```
/* ------------------------------------------------
   User Completed data structure
 * ------------------------------------------------ */

typedef struct Lst
    {
      float Start;  /* start of section to integration */
      float End;    /* end of section                  */
      float Radius; /* of circle being integrated      */
    } Conquer;      /* Skeleton defined naming         */
```

Figure 16 - Data Structure input for Integration example.

[7] This example may be implemented using the Task Queue skeleton, but is used here to illustrate the working of the Divide and Conquer Skeleton.

User Input

Data Structure

As a simple, single-record data structure was used (see Figure 16), the block read and writes provided by the skeleton can be used.

To integrate the space under the curve it is broken up into a number of small integrals. The start position of the rectangle must be kept in order to determine the height to the curve at that point. The end position of the interval must also be held to determine the current width, and, from this, whether the interval has reached its trivial size. In the circle example of Figure 17, the theorem of Pythagoras is used to calculate the height at a point.

Trivial

With all integration examples the trivial case returns true when the base of the rectangle under the curve had reached a predetermined minimum size.

SimplySolve

The solution to the simplest case is the area of the rectangle under observation. This is the only user code fragment which is specifically written to solve for the area under a circle. The formula used can be replaced to calculate the area under any curve.

Decompose

This code fragment must be written to return two sub-problems, each to solve part of the area of the parent.

CombineSolutions

This code fragment adds the two areas returned from its children processes.

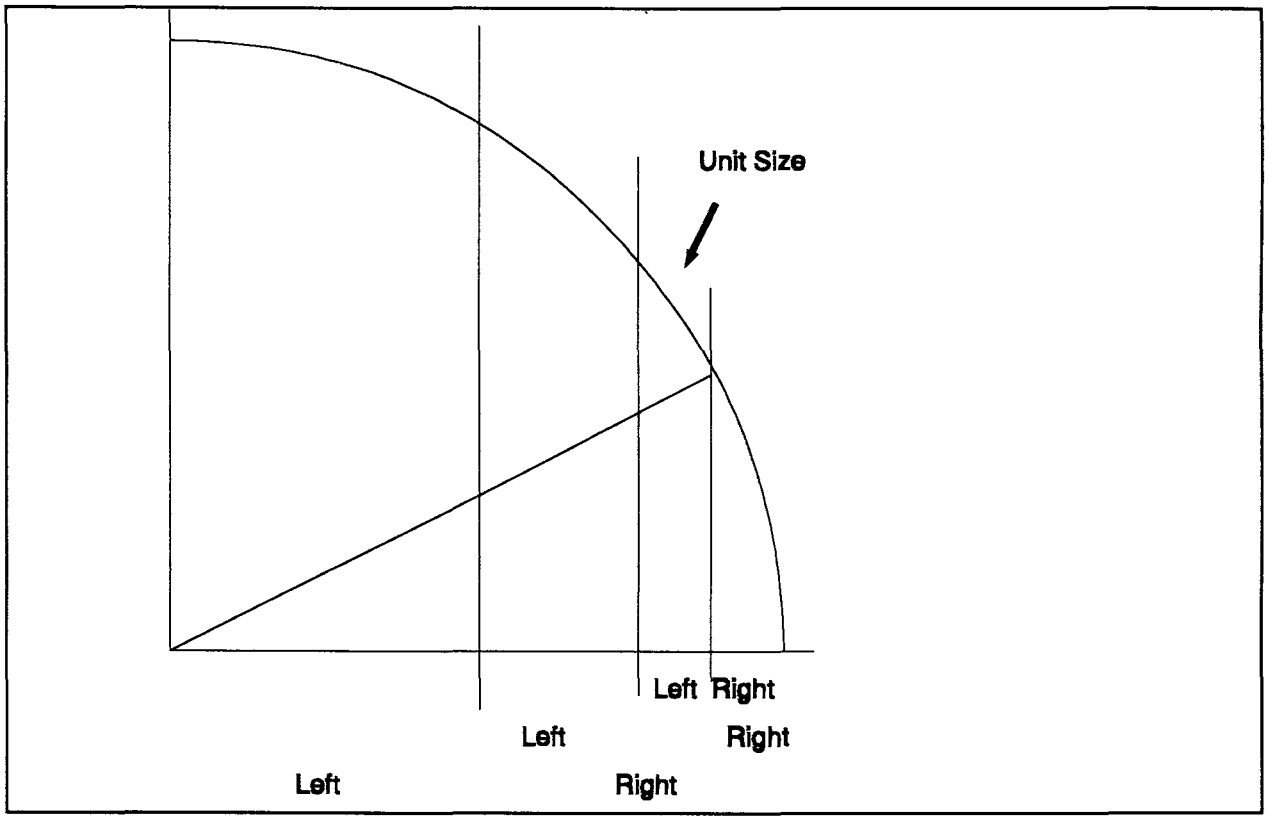

Figure 17 - Determining the area under a Circle.

4.9.2 Nfib

Nfib calculates the number of fib calls made to generate a solution for the Fibonacci series.

User Input

Data Structure

A simple data structure was used (see Figure 18). This communicates the nfib value for which a solution is sought, as well as providing space for the eventual solution.

Trivial

The trivial case is reached when the nfib values for 1 or 2 are needed. Trivial returns true for these values.

SimplySolve

This routine will return the solutions to the simplest cases. The nfib of 1 or 2 returns 1.

Decompose

This procedure will return two new sub-problems. For example the nfib(30) will return the sub-problems of nfib(29) and nfib(28).

CombineSolutions

This routine adds the values of its two sub-problems to reach a solution.

```
/* ------------------------------------------------------
    User defined data structure
 * ------------------------------------------------------ */

typedef struct Lst
 { float FindFib;   /* Nfib value to be calculated */
   float Solution;  /* Solution to Nfib wanted     */
 } Conquer;         /* Skeleton defined naming     */
```

Figure 18 - Data Structure input for the Nfib example.

4.9.3 Sorting

Sorting a list of length N may be done using a Quicksort algorithm. The algorithm works by recursively partitioning the data, with all the elements in the one partition smaller than those in the other. Once the base case is reached, all the sub-problems are concatenated to form the problem solution.

User Input

Data Structure

If a simple data structure is used, as in the integration example above, the whole user structure may be communicated as one block transfer. If a more complex data structure is needed, such as a link list, (see Figure 19), the data structure required as input by the skeleton must contain a pointer to the first element in this link list. Users must pack their data into a contiguous block of memory when using the skeleton block communication. A child will have to read in the block and recreate the link list. This is essential for efficient data transfer.

Trivial

In a Quicksort routine the trivial case is reached when the list to be sorted consists of one data item only.

```
/* ------------------------------------------------
 User defined data structure
 * ------------------------------------------------ */

typedef struct Sort
        {
          char Thing;         /* type of element to sort */
          struct Sort *Next; /* pointer to next element */
        } Slist;

typedef struct
        {
          Slist *L;           /* pointer to first element */
        } Conquer;
```

Figure 19 - Data Structure for Quicksort Example.

SimplySolve

The solution to the simplest case is simply to return the trivial list of one data item.

Decompose

A comparator must be chosen and used to partition the list. All elements bigger than the comparator must be placed in one list and all others in another.

CombineSolutions

As this is a Quicksort, the solution list is formed by concatenating the solution lists of the children. As a result of the decompose function, all the elements in the left solution list will be smaller than those in the right list.

4.9.4 Strassen's Matrix Multiplication

The problem consists of multiplying two N by N matrices, A and B, to yield a solution matrix C. The solution may be determined by dividing A and B into four sub-matrices[AHO83] and computing the corresponding four matrices for C, as shown in Figure 20. This is done by eight independent multiplications, followed by four additions. The eight multiplications may be calculated in parallel.

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

$$\begin{vmatrix} A_{11} \cdot B_{11} + A_{21} \cdot B_{12} & A_{12} \cdot B_{11} + A_{22} \cdot B_{12} \\ A_{11} \cdot B_{21} + A_{21} \cdot B_{22} & A_{12}^{12} \cdot B_{21} + A_{22} \cdot B_{22} \end{vmatrix}$$

Figure 20 - Formula for calculating the product of two Matrices.

User Input

In order to simplify the user inputs to the skeleton routines, a library of functions such as Split, Print, Free, Copy, Join, Fread, Alloc, Add is used. Figure 21 gives their type definitions. Implementing the user code for the skeleton is much simplified with these routines defined.

Data Structure

The data structure Conquer has pointers to the two matrices to be multiplied, as well as to a result matrix (see Figure 22).

Trivial

The trivial case is reached when two 2 by 2 matrices must be multiplied.

SimplySolve

The trivial case may be solved using the formula of Figure 20. The solution is placed in a new matrix pointed to by Result.

Decompose

Split both input matrices into four sub-matrices. The formula defines which of the new matrices from the original two must be paired as sub-problems to be multiplied.

CombineSolutions

Add corresponding matrices in accordance with the formula, and to form a new result matrix.

```
/* Take a matrix returning its four quadrants as new matrices */

   void   MatrixSplit(Matrix *M, Matrix **Q1, Matrix **Q2,
                      Matrix **Q3, Matrix **Q4);


/* Print a matrix */

     void MatrixPrint(Matrix *M);

/* free a matrix, given it a pointer to it */


     void MatrixFree(Matrix *M);


/* Copy a portion of a matrix from an element start to an
   element end, to position in another matrix */

     void MatrixCopy(Matrix *FM, Matrix *TM, int Fstart, int
                     Tstart, int Row, int Column);


/* Place four matrices as the four quadrant of a new matrix */

     Matrix* MatrixJoin(Matrix *M1, Matrix *M2, Matrix *M3,
                        Matrix *M4);


/* Read a matrix from file */

     Matrix *MatrixFRead(char *Name, int Row, int Column);


/* allocate space for a specific size matrix */

     Matrix *MatrixAlloc(int M, int N);


/* add two matrices and return a third as a result */

     Matrix *MatrixAdd(Matrix *M1, Matrix *M2);
```

Figure 21 - Definitions of the Operations on Matrices.

```
/* ------------------------------------------------
   User Completed data structure
   ------------------------------------------------ */

typedef struct Mtrix
   {
   int Row, Column;  /* Matrix Size   */
   int Data[1];      /* First Element */
   }Matrix;

typedef struct
     {
       Matrix *M1; /* First Matrix  */
       Matrix *M2; /* Second Matrix */
       Matrix *Rt; /* Result Matrix */
     } Conquer;
```

Figure 22 - Data Structure for Matrix Multiplication Example.

4.10 Chapter Review

This chapter introduced the concept of a Divide and Conquer Skeleton. An example MaxMin, which finds the maximum and minimum of a list of integers, was presented. It was used to illustrate the development of the skeleton. Code common to all divide and conquer problems was extracted to form the skeleton, as well as the code fragments which form user input to the skeleton. Types of the user code fragments and the skeleton were shown.

The chapter went on to discuss the way in which the skeleton dealt with the tricky issues of code distribution, communication, synchronization, and termination. A distribution algorithm was developed around a list of constraints. It can be decoupled should a new configuration be required. Skeleton controlled communication structures were provided to allow user data structures to be passed between processes. Four examples which illustrate the use the Divide and Conquer Skeleton were presented. They will be evaluated in chapter 7.

Chapter 5

5. The Task Queue Skeleton

The concept of a task queue is often useful when dealing with a large job which may be broken up into small sub-problems, but in an irregular manner. The smaller jobs may or may not be interdependent. This means that a number of independent processors can tackle the load. A job queue is initialised with a problem, its first job. This task is issued to a worker which, while processing the task, may return any number of smaller sub-tasks that other processes can operate on. Eventually a final result is returned. A well known example which may be implemented using the task queue paradigm is the eight queens problem (see applications section 5.8).

5.1 Task Queue Algorithm

A central feature in the development of a Task Queue Skeleton is the structure to implement the queue of jobs to be processed. In this implementation there will be a single central copy of the queue on a master process (see "distribution" section 5.3 for a discussion on queue placement schemes). Problems or tasks will be written to workers, and solutions written back, via a load-balancer. In this way, work will be evenly distributed across all idle workers.

The master processor, which is responsible for maintaining the queue, operates according to Figure 23.

Workers will be distributed across the transputer network. Their function is to read in problems, operate on them, and write new sub-problems or solutions back to the master. Any sub-problems generated will be placed on the task queue. It should be noted that all results passed back from the worker could represent a partial, or complete solution. It is the task of the master to determine whether this result is a partial or full solution. A worker will operate according to the algorithm of Figure 24.

```
M(x) =

Initialize the Queue      (with one or more problems)
Repeat in Parallel
 Write Job(s)             (to workers)
 Read Job or Answer       (from workers)
Until Answer
```

Figure 23 - Task Queue Master's Code.

```
f(x) =

REPEAT
 IF  task available THEN
     try to grab a task (from the master)
 IF successful THEN
     BEGIN
          execute task
          send any tasks created, or solutions to the Master
     END
UNTIL no task is available and all processors are inactive.
```

Figure 24 - Task Queue Worker's Code.

In specifying a problem for execution by the Task Queue Skeleton, a user must provide:

- JobDataType - A type specification for the data structure used to transmit jobs. This data structure is also used to make up the task queue, which will vary depending on the particular problem.
- Queue Initialization - An initial problem instance for the master process. The first worker will then generate sub-problems for other workers, which will be placed on the task queue. Queue Initialization returns problem list of type α.
- Problem Solver - This procedure describes the operations a worker must perform on the data to reach a solution. Given a problem of type α, Problem Solver will return a full solution, or a list of partial solutions to be processed further, both of type $[\alpha]$. The master will determine whether a full solution has been reached.

- Queuing Discipline - This takes the form of a parameter to set some options such as the queuing discipline and maintenance of the queue. Although this implementation works on a LIFO basis, some user problems may operate more efficiently with a different queue discipline. A number of queuing methods can be programmed and made available to the user.
- Answer - An Answer routine to process all answers received from the workers. This enables the master process to determine the final solution (full solution) from all the sub-solutions (partial solutions) found. If data from a worker is not an answer, it is another problem which must be added to the queue. Answer takes a list of type α and may return a solution of type ß.

```
TQSkeleton : is a function which takes four arguments

  QueueInitialization : [α]
  ProblemSolver       :  α  -> [α]
  Answer              : [α] ->  ß

and it returns a new function
  TaskQueue : α -> ß


In all these signatures, α stands for the type of the
problem, ß for the type of the solution.
```

Figure 25 - Function Inputs to the Task Queue Skeleton.

The types for these inputs are illustrated in Figure 25 and the type of the Task Queue Skeleton in Figure 26.

```
TaskQueue : [α]            /* QueueInitialize */
            (α → [α])      /* ProblemSolver   */
            ([α] → ß)      /* Answer          */
            α              /* Problem of type α   */
          → ß              /* Solution of type ß */
```

Figure 26 - Type specification for the Task Queue Skeleton.

5.2 Solution Development

The Task Queue introduces an explicitly parallel style of Skeleton. It has a task queue which is initialized with an initial problem which requires processing. The skeleton then issues each of the available workers with a problem from the queue. The worker operates on the problem it has received, and returns a solution or a new problem to be added to the process queue. If the problem a worker is processing depends on the solution of a another sub-problem, it is replaced on the queue until the solution to the sub-problem is found. All the processors execute in parallel so no assumptions may be made regarding their order of evaluation.

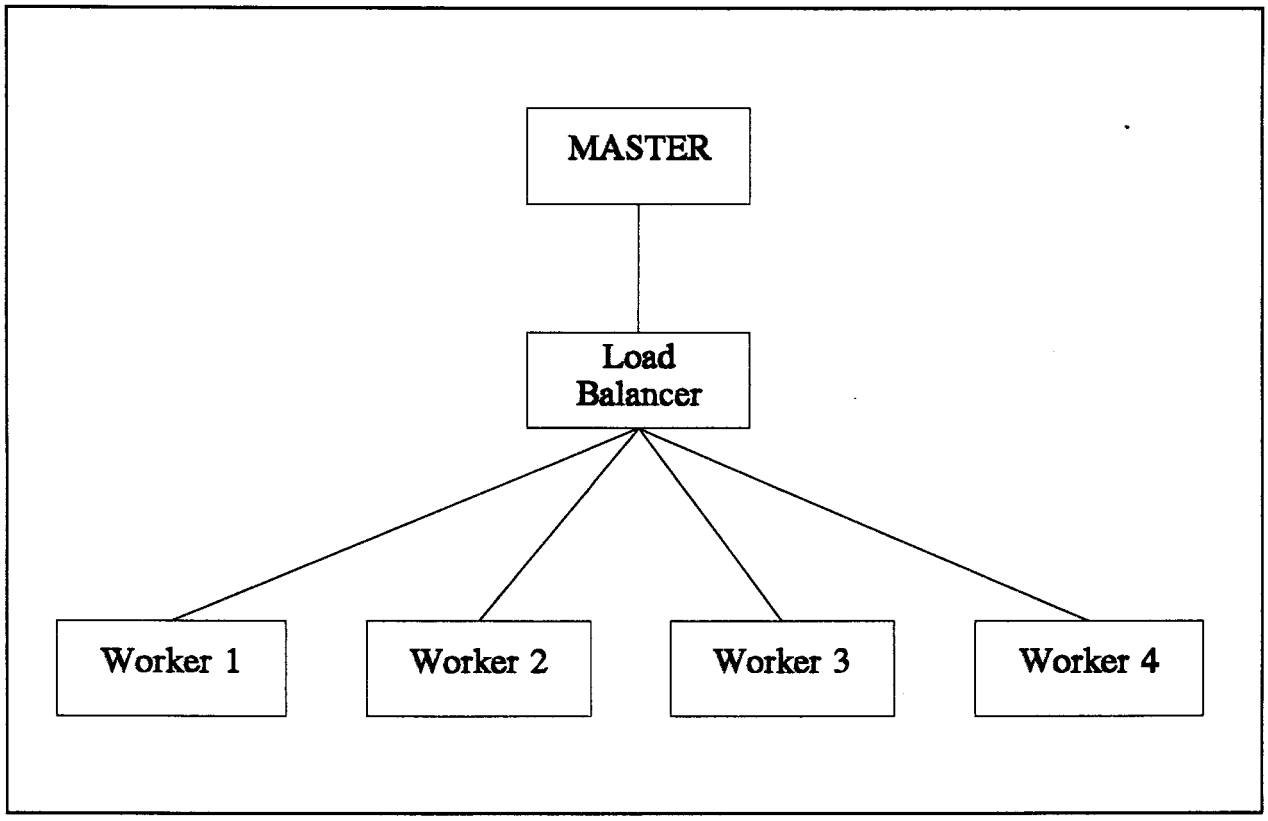

Figure 27 - A Farm of Workers.

The skeleton is based on the notion of a farm of workers, each worker doing a piece of a greater problem. A load balancer is used to control communication between the master and the workers (see Figure 27).

The skeleton handles the otherwise difficult problem of queue maintenance such as adding problems to the queue, removing work from the queue and allocating problems to workers.

5.3 Distribution

The location of the queue is dependant on the type of hardware available. In a shared memory system the implementation would be simple. However, in the local memory model of the transputer based system, a number of options regarding access to the queue must be considered. Either the queue must be kept on a single identified transputer or full or partial copies must be made available on all workers. All of these methods will introduce further communication overheads.

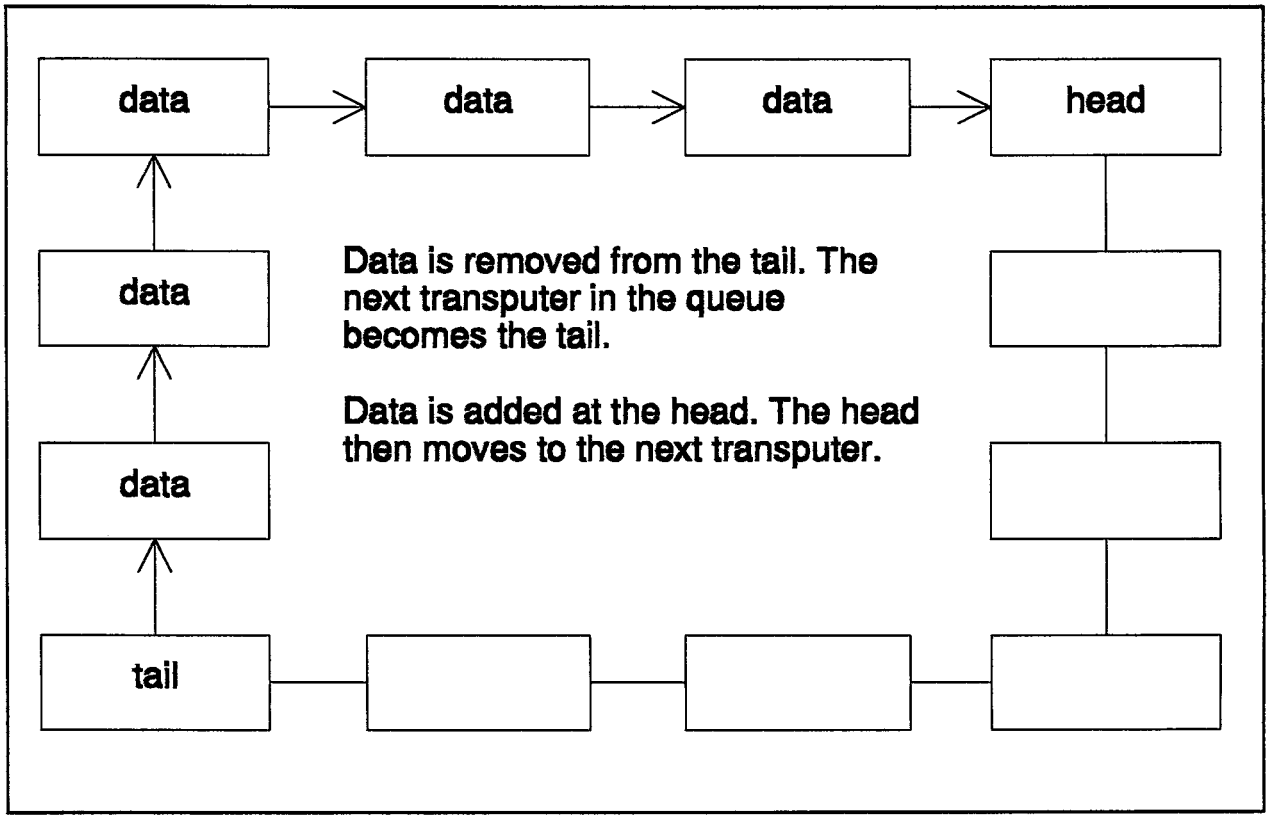

Figure 28 - Cole's Task Queue.

Cole suggests a system whereby each transputer holds part of the queue (see Figure 28). A centralized approach is vulnerable to an idealised time step, when there is no shared memory, and when implemented

on a large number of processors. All processes are uniquely numbered, and one of the transputers will be identified as the tail of the queue and another as the head. The algorithm works in two phases. During the first phase all processes with tasks to send to the queue generate a personal offset from the queue head. Tasks from all processes are then sent to (processor number head - offset) queue position, in parallel and the head is updated. During phase two all processes requiring a task generate an offset from the tail. Communication takes place in parallel and the tail is moved.

The implementation of this thesis employs a simpler algorithm. The entire queue can be placed on a single transputer. This transputer can also process all results received. All other transputers will act purely as workers. The load balancer will read tasks from the queue and pass them to available workers. It will also read results, and tasks created from the workers, and pass them to the queue on the master process.

Implementation

Each worker should ideally be connected directly to the master and its job queue. For more than four worker processors and one master, this is not possible in the transputer environment where there are four links per transputer. The optimum is to connect the workers in such a manner that as many as possible (maximum of four) are connected to the master. Referring to the hardware configuration diagram of Figure 2, transputer 1 is connected to the output (to the host computer) and only has three available connections. Transputers 2 to 8 are all connected to four other transputers. Transputer 2 is selected for the master. Four transputers are connected to it (1, 4, 6 and 7). To make data available to workers on transputers (3, 5 and 8) not directly connect to the master requires routing through a maximum of one transputer. Helios will handle this.

One restriction with the standard CDL load balancer is that it expects only one result from a worker for every one problem sent to it. In some cases a worker of the Task Queue Skeleton may send no solutions, or it may send many solutions. The load balancer was modified to allow this (see Appendix A).

5.4 Communication

The load balancer allows communication, using packets, between the master transputer and the workers. A standard size header is sent detailing the length of the packet. The skeleton uses this information to read in the data. All communication can therefore be handled by the skeleton. The user procedure ProblemSolver receives the data structure and can work on it directly. As with the Divide and Conquer Skeleton, the user is required to define this data structure as part of the input into the skeleton. When writing problems out to the workers, the skeleton knows the size of the message because a user defined data packet is used. When reading problems from a worker, messages of different lengths may be received. The worker must therefore write out a header. The master receives this header and is then able to read a message of the correct length. The load balancer does not have access to the user data structure definition and thus does not know the size of the message it must transmit. It also reads a standard header to determine the size of the message to be passed.

5.5 Synchronization

Access to the task queue is controlled by the load balancer. The master polls its communication channel from the load balancer and reads in results as they are passed. It writes available work to the load balancer for processing.

5.6 Termination

Termination occurs when the job queue is empty and there are no outstanding tasks executing on any worker (The skeleton is able to calculate the number of outstanding tasks as each worker automatically sends an end-of-task header to the master). Termination is completely controlled by the skeleton. A termination header is sent to the load balancer by the master, which in turn broadcasts it to all workers, and then terminates itself.

5.7 The User Interface

The input required for this skeleton is simpler than that required for the Divide and Conquer Skeleton. The user must supply three simple procedures. The processing of the interface is the same as for that of the Divide and Conquer Skeleton.

5.8 Applications

Greedy algorithms (see example below) such as those which may be solved using the Divide and Conquer Skeleton are not suitable when solving problems for which the best solution is not immediately apparent. Dynamic Planning techniques[HAR87] are needed to make more subtle choices. The shortest path problem illustrates this difference.

5.8.1 Shortest Path

Shortest path solutions to Figure 29 can be found using the two techniques:

- A Greedy Algorithm A -> C-> F-> H = 15
- Dynamic planning A -> D -> H = 12

The greedy algorithm solves the problem by following the path that is immediately shortest. This results in it finding a non-optimum solution. It can be seen that dynamic planning is needed here to find the optimum solution. The Shortest path Algorithm is an example of dynamic planning. At each step the algorithm delays making a decision until its can be sure which will provide the best solution. In order to arrive at the solution above, the shortest path algorithm carries out the following steps.

The shortest path from A to H is the cost from A plus the minimum of the shortest paths from B, C or D to H.

Therefore Length(A) = min (5 + L(B), 3 + L(C), 10 + L(D)).

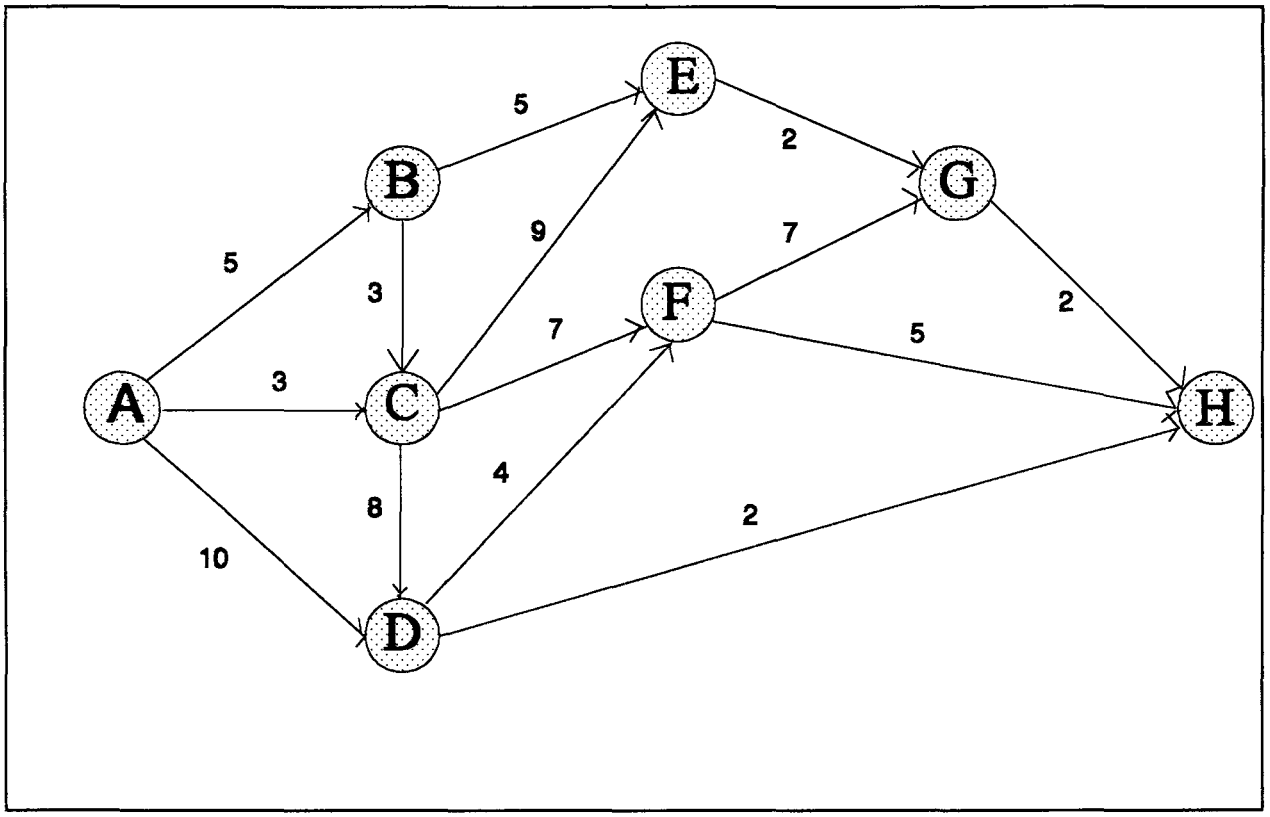

Figure 29 - Shortest Path Problem.

In turn

Length(B) to H = min (5 + L(E), 3 + L(C))

Length(C) = min (9 + L(E), 7 + L(F), 8 + L(D))

....

Length(F) = min (7 + L(G), 5 + L(H))

....

Length(H) = 0

In finding the solution from A to H the algorithm delays the decision on taking route B, C, D until it can determine which is the shortest path to H.

Problems such as the example to find the shortest path through a graph are best implemented using the Task Queue Skeleton, as apposed to the Divide and Conquer Skeleton which employs a greedy algorithm.

The task queue will build up the following solutions.

Task Queue	Solution	path
L(A)	12	5 + L(B), 3 + L(C), 10 + L(E)
L(B)	11	3 + L(C), 5 + L(E)
L(C)	10	9 + L(E), 7 + L(F), 8 + L(D)
L(D)	2	4 + L(F), 2 + L(H)
L(F)	5	7 + L(G), 5 + L(H)
L(H)	0	0
L(G)	2	2 + L(H)
L(E)	4	7 + L(G), 5 + L(H)

The shortest path problem was not implemented, but is used here to illustrate the difference between the Task Queue and Divide and Conquer Skeleton.

5.8.2 Factors

Given a range of numbers, the program determines which number in the range has the most factors. The master is initialised with the full range. This is broken down into a number of smaller ranges which are placed on the job queue, to be issued to workers. Each worker determines which number in their specific range has the most factors. This result is then passed to the master, who compares solutions and determines the final result. In this way many smaller ranges are computed in parallel.

User Input

Data Structure

In this example a worker must receive a number indicating the start of the range of numbers it must process and how many numbers are in that range. It must also send back the solution integer and its number of factors (see Figure 30).

```
/* ------------------------------------------------------
   Communication structures to communicate full and partial
   solutions to the Master for evaluation
 * ------------------------------------------------------
 */

#define Base  Count

typedef struct UserJob
   {
     int Count;  /* Start of the Range
                    or Count for Best Factor */
     int Best;
   } UserJob;
```

Figure 30 - Task Queue data structure for the Factors Example.

Init

Initialises the jobs queue with a number of jobs. Each job will contain a range of numbers in which the one with the greatest number of factors must be found.

Answer

The answer routine will receive all potential solutions and determine the final result.

Process

This is a simple routine which, given a range of integers, will return the one with the most factors, along with the number of factors.

5.8.3 Queens

This is a well known problem where a chessboard of size n by n must be set up with n queens such that no queen attacks any other. The task queue is initialized with an empty board. All possible combinations for the safe placement of queens in the first column are calculated. (In the case of an empty board it is possible to create n tasks. Each will have a queen in the first column, but in consecutive rows). These combinations are all returned as separate incomplete tasks to the master. These tasks are placed on the

job queue to be issued to workers, which attempt to find all combinations for placement of other queens in the next column. If any attempt fails to find a safe placement, that board is discarded.

User Input

Data Structure

An array represents a chess board. A one dimensional array is sufficient because only one queen will be placed in a column. The first element is a count indicating how many queens have been placed (see Figure 31).

```
/* ------------------------------------------------------
   Communication structures to communicate full and partial
   solutions to the Master for evaluation
 * ------------------------------------------------------
 */

const int Size = 8;       /* size of the Board   */
typedef int UserJob[8];   /* array to hold Board */
```

Figure 31 - Data Structure for the Eight Queens Problem.

Init

The job queue is initialised with an empty board. No queens have been placed. The first worker will create new jobs with queens in the first positions on the board.

Answer

A solution is reached when all the queens have been placed.

Process

A worker receives a partially completed board and finds all possible solutions for the next column, these are then sent to be placed on the job queue.

5.9 Chapter Review

This chapter introduced the concept of a Task Queue Skeleton. An important decision was the placement of the task queue on a single transputer. Algorithms were described for the master process, which maintains the task queue of jobs, and for the worker processes, which execute the user processes. The types of the inputs, JobDataType, QueueInitialization, ProblemSolver, Queuing Discipline and Answer, required by the skeleton were detailed. The complex issues of distribution, communication, synchronization, and termination were discussed, in relation to the implementation of the Task Queue Skeleton. Two distribution algorithms were presented. Details and an implementation were shown for one of the algorithms. This algorithm made use of a load balancer to control the communication of tasks between the master and the workers.

The shortest path problem was discussed to contrast the Divide and Conquer Skeleton and the Task Queue Skeleton. It was used to show that problems for which the solution is not immediately apparent, are better suited to the Task Queue Skeleton and the dynamic planning it employs. Two applications illustrated the use of the Task Queue Skeleton and will be evaluated in chapter 7.

Chapter 6

6. Reuse of Skeletons

A lot of the methods for implementing algorithmic skeletons have being criticized for their lack of flexibility[FOS90a]. It is frequently difficult to find the optimum skeleton for a user problem, because it must be written for one specific skeleton. User choices within a skeleton are limited. To a large extent, issues such as physical code distribution are hard-wired into the skeleton. It is not possible to alter the skeleton code, or to build new skeletons on top of existing ones.

These issues are addressed through the use of reduced parallel algorithmic skeletons. This is the central concern in the approach to the concept of algorithmic skeletons proposed by Foster and Stevens in their paper "Parallel programming with Algorithmic Motifs"[FOS90a]. The mechanisms by which they accomplish reuse are those of "modification" and "composition". Modification allows a skeleton to be redefined with a different mapping algorithm, whilst composition allows the features of a skeleton to be combined with another skeleton to form a new skeleton. These mechanisms are possible through the use of high level languages and source-to-source transformations. The aim of this research is to allow sequential programmers access to parallelism, and to allow existing programs to be altered to work using an algorithmic skeleton. Therefore different methods had to be found to implement reuse.

Figure 32 shows Foster's idea of an algorithmic motif. In the diagram a generic input routine is connected with a motif from a library.

6.1 High Level Languages

Foster and Stevens required that their motifs (skeletons) be implemented using a high level language, such as the concurrent programming language Strand[FOS89]. They believe that low level languages such as 'C' are write-only and therefore don't provide an efficient medium to archive expertise. Programs written

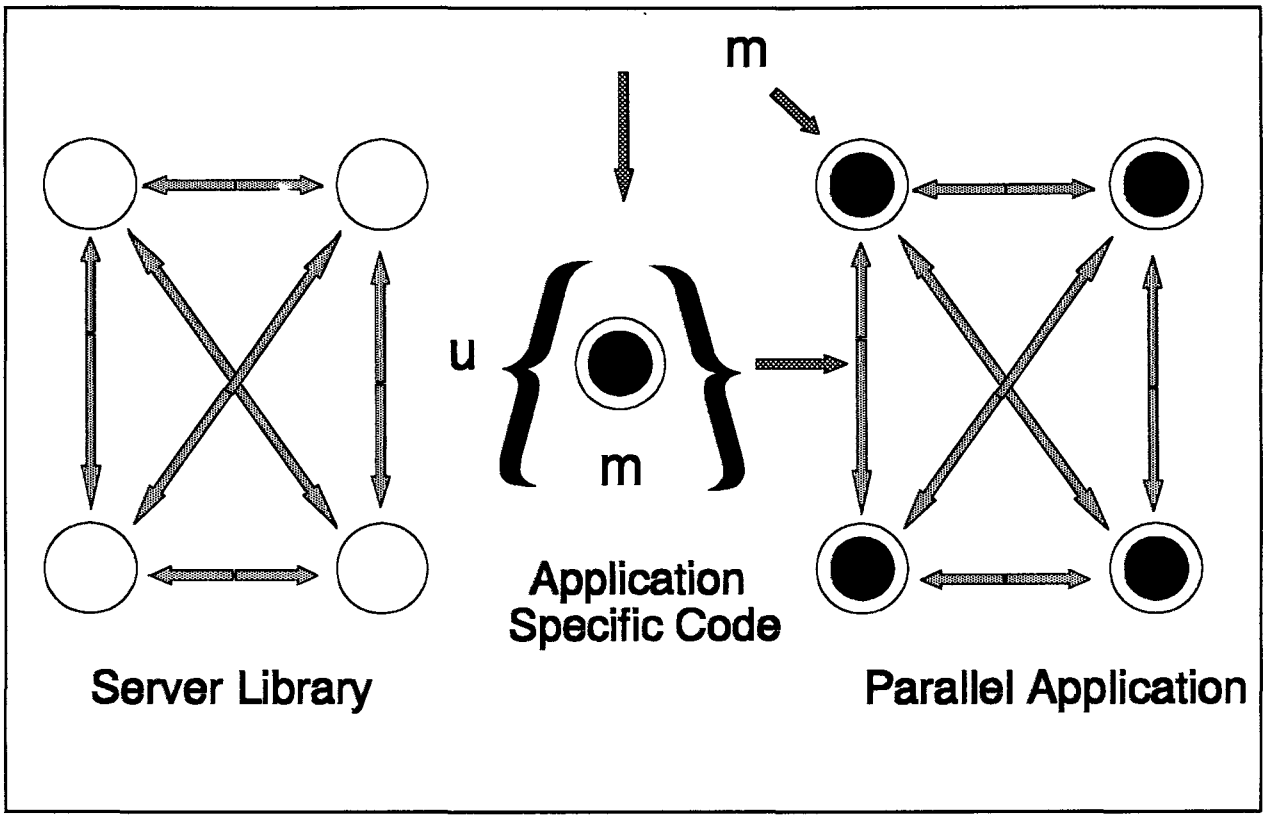

Figure 32 - Foster's Idea of a Algorithmic Motif.

in a high level language would also be more modular and portable. The issue of inefficiency is addressed by implementing computationally intensive code at a low level.

```
If A is an application program and
   M = {T,L} is a motif, then
   the application of M to A yields a new program
   A' = M(A) = T(A) union L.

Where a motif is denoted by {T,L}, T is a transformation
and L is a library.

The implementation of a motif comprises both a
source-to-source transformation and a library program
```

Figure 33 - Foster's Motif Transformations.

6.2 Transformations

Modifications

Transformations to allow a user problem to be implemented using different skeletons are termed modification. The selection of an optimum skeleton may depend on the type of input data to the skeleton application. One skeleton may be more efficient with balanced data, whilst another may be able to solve a problem with skewed data more efficiently. In Foster and Stevens' work, transformations are implemented by inserting macros into the user code to control, for example distribution. A transformation module is written as part of the skeleton implementation (See Figure 33). The macros may be expanded according to the motif being used. In Cole's implementation, the user is required to insert skeleton specific code to control communication. This renders the user application code skeleton specific. Using reduced parallelism algorithmic skeletons, no skeleton specific code is required. The user code is thus not tied to a particular skeleton and may be tested with any suitable skeleton.

Given an existing motif $M_1 = \{T_1, L_1\}$

Rather than implementing a new motif M from scratch, we can implement a simpler motif $M_2 = \{T_2, L_2\}$ that composed with M_1 provides new functionality.

M = M_2 compose M_1 and
$M(A) = M_2(M_1(A)) = T_2((T_1(A) \text{ union } L_1) \text{ union } L_2)$

Motifs allow the development of new motifs by composition.

Figure 34 - Foster's Motif Composition Function.

Composition

Transformations to allow a user to build skeletons from existing skeletons are termed composition. This can be a very useful technique allowing greater flexibility in the use of skeletons (see Figure 34).

6.3 Reuse of the Skeletons of this Thesis

The integration example implemented under the Divide and Conquer Skeleton (section 4.9.1) could possibly be implemented more efficiently under the Task Queue Skeleton. Under the implementation of this thesis, reworking of the user application would be required to allow this. A standard interface to the skeletons, coupled with Foster's idea on transformations, would make it possible for a user application to be tested with a number of different skeletons. In this way a user will be able to determine the most efficient skeleton for a particular application.

As the skeletons in the work of this thesis have been developed in a very modular form, and user applications include no skeleton specific code, it is possible for a user to simulate Foster and Stevens' concept of composition in terms of user application programs. For example, if a skeleton application is written to determine the area under a circle, it could be used as a base to create a skeleton application to find the area under a different curve. This form of reuse is not in the spirit of Foster's work.

These are useful areas of future research.

Chapter 7

7. Evaluation of the Skeleton Approach

There are two major problems associated with implementing skeletons on a physical multi-processor architecture[RAB90].

- Tasks that are too tiny in comparison to the communication overheads they generate will execute faster sequentially.
- If too many tasks are generated by a problem, the systems resources may be overwhelmed.

7.1 Divide and Conquer

Parallel execution of f(x) in Figure 6 has the obvious scheduling strategy of allocating a processor to each task in the system[RAB90]. The approach has three important drawbacks:

- Inactive Processors - The processors executing the interior nodes of the tree will be inactive for most of the time.
- Exploding Computational Tree - The computational tree will become much larger than the physical machine.
- Granularity Problem - The fine granularity will cause excessive communication overheads.

The distribution presented in the work of this thesis solved the first two shortcomings (see section 4.4). However, the approach still suffers from too fine a granularity when solving problems requiring large amounts of data.

A solution to this, suggested by Rabhi et al. [RAB90], consists of a system which performs run-time complexity analysis on the problem. Tasks are not created once a threshold for minimum grain size is reached. The result is a coarser grain. They refer to this scheme as Dynamic Partitioning. The factors that influence Dynamic Partitioning are the number of processors, the load balancing strategy, and the nature and complexity of the problem. This is the antithesis of the Throttle approach, in which problems are decomposed to the maximum extent of the hardware, as used in most skeleton implementations reviewed [RAB90].

Three styles of problems are identified by Rabhi et al.:

- Regular Balanced - The problem divides equally, resulting in an equal load on each sub-tree. Examples include a program to find the sum of a list of numbers, a program to do matrix multiplication and a program to find the number with the most factors in some range.
- Regular - The size of the sub-problems can be predicted but they don't divide into equally. For example a program to generate the Fibonacci series.
- Irregular - The size of the sub-problems cannot be predicted, and are unbalanced. For example a program to perform a Quicksort.

Rabhi et al. show that an optimum partition exists for every problem. Regular Balanced problems require a system which breaks into exactly the number of available processors. Regular problems, that are not necessarily balanced, are best solved using smaller tasks with load balancing. Irregular problems are data dependant and a general solution is not possible.

User tuning of applications is contrary to the goals of algorithmic skeletons because it requires skilled input. However it is possible to implement some tuning in the algorithm presented in this thesis. For example, it is possible to alter the problems grain size and the process load.

The results produced by the Divide and Conquer Skeleton of this thesis will be discussed in terms of the problem classifications above. Four virtual tree topologies have been defined to solve problems on trees

of depth 1, 2, 3 and 4. This corresponds to trees of 1; 2, 4 and 8 processors. The grain of a problem can be increased by reducing the depth of the processor tree.

7.1.1 Regular Balanced Problems

The implementation of an integration problem showed that the standard skeleton configuration will scale well for regular problems, with limited communication requirements (see Figure 35).

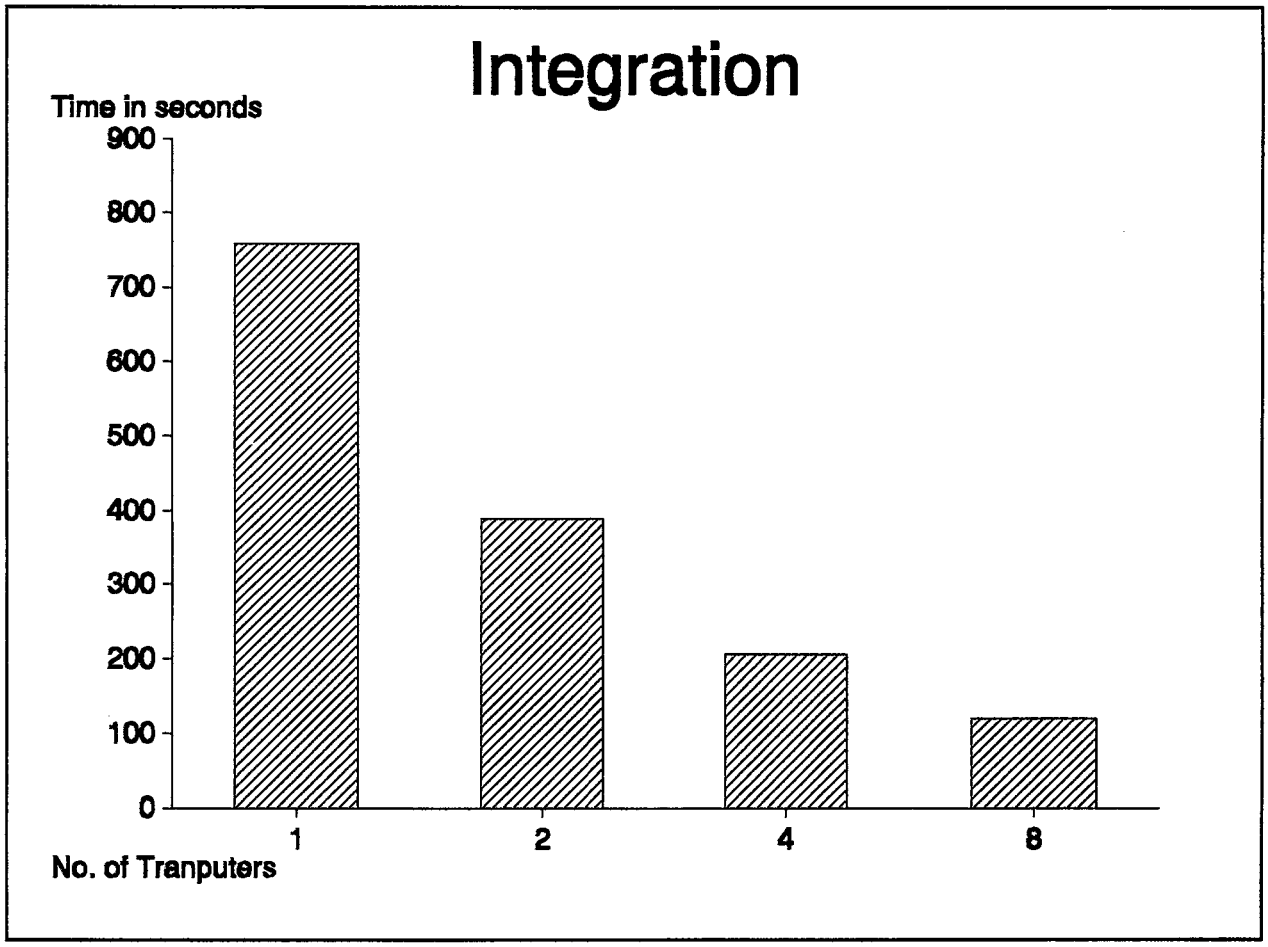

Figure 35 - Speed up of integration example.

When communication becomes an important part of the problem, the speed up with increased processor numbers is reduced, but still good. Multiplying two matrices makes intensive use of communication links.

This problem is suited to execution under this thesis's Divide and Conquer Skeleton, but its performance is offset by the heavy communication load (see Figure 36).

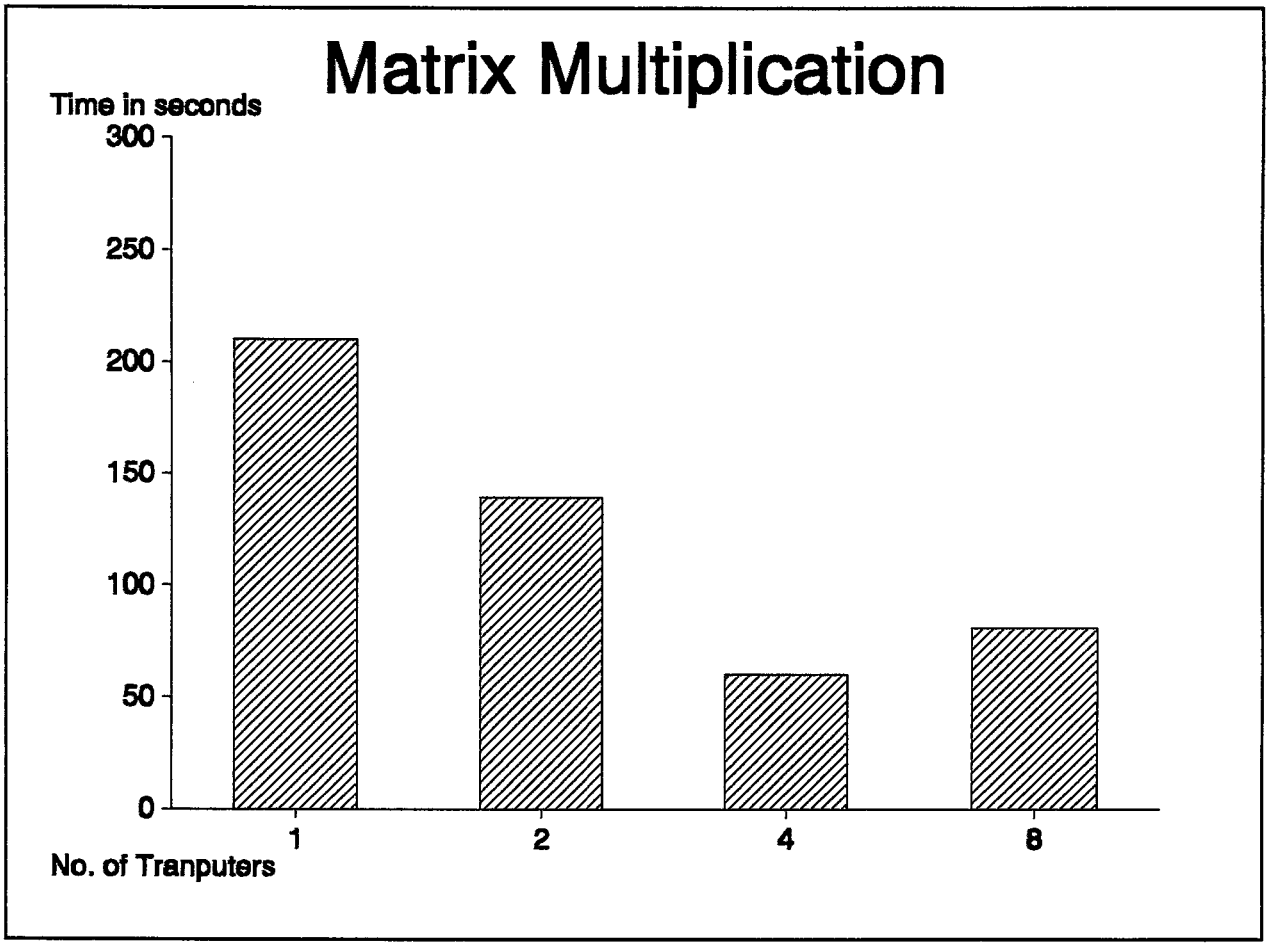

Figure 36 - Changing execution speed of the Matrix Multiplier.

When the cost of communication exceeds that of computation, performance begins to decreases. The graph of Figure 36 is predicted in the paper by Rabhi et al., in that performance tails off. The marked improvement in speed between 2 and 4 processors is interesting. A number of factors could be responsible:

- An ideal processor to communication ratio may occur with 4 transputers.
- Although the tree size increases, there are only 2 more interprocessor links.

- The most likely explanation is that the depth of the tree passes through an optimum value, as is suggested in the paper by Rabhi et al.

Figure 37 shows $\log_2$ (time) for the matrix multiplication.

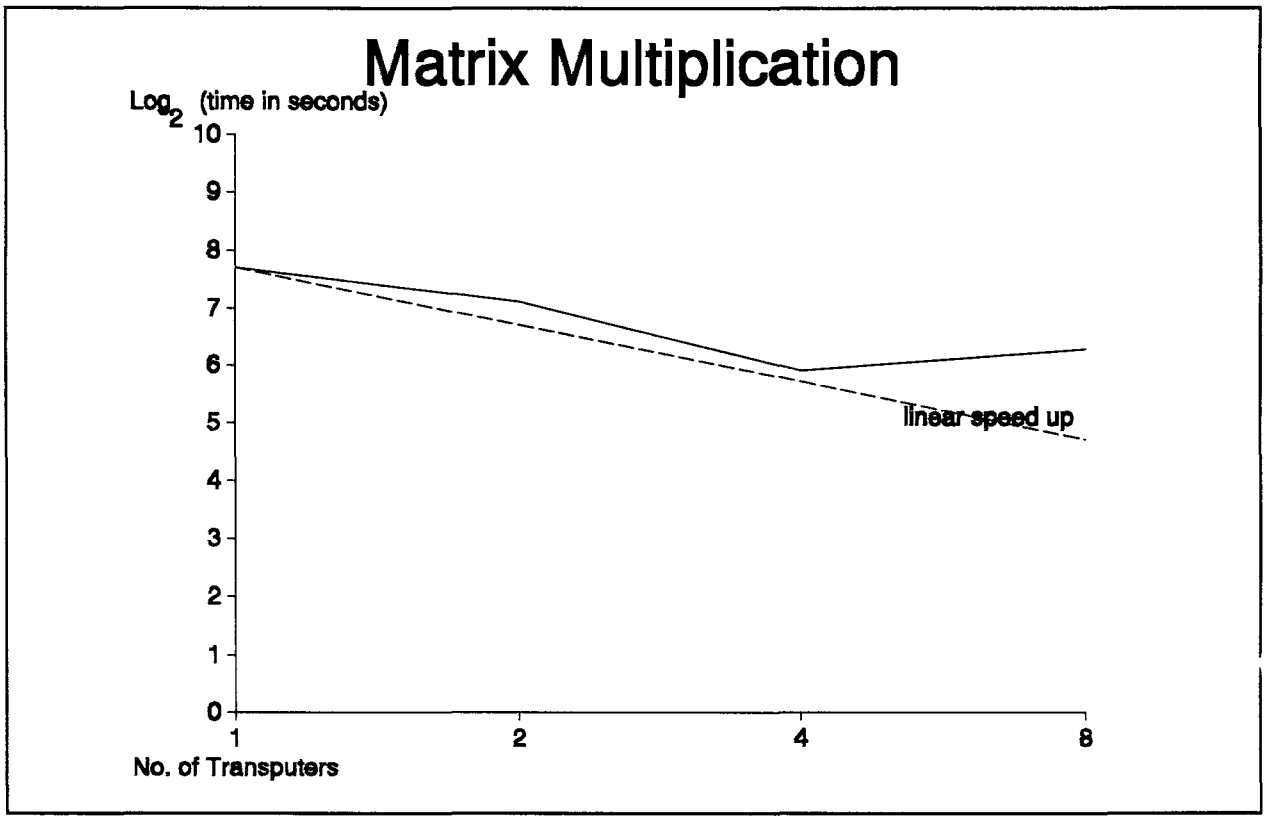

Figure 37 - $\log_2$ time for Matrix Multiplier.

7.1.2 Regular Problems

A skeleton application to generate the Fibonacci series was written to illustrate performance of Regular problems (see Figure 38).

It can be seen from the graph that performance increases significantly as the number of processes doubles from 1 to 2. This increase in performance tails off as the number of processes increases beyond two, as a result of the unbalanced nature of the problem's decomposition. This example can be compared to the

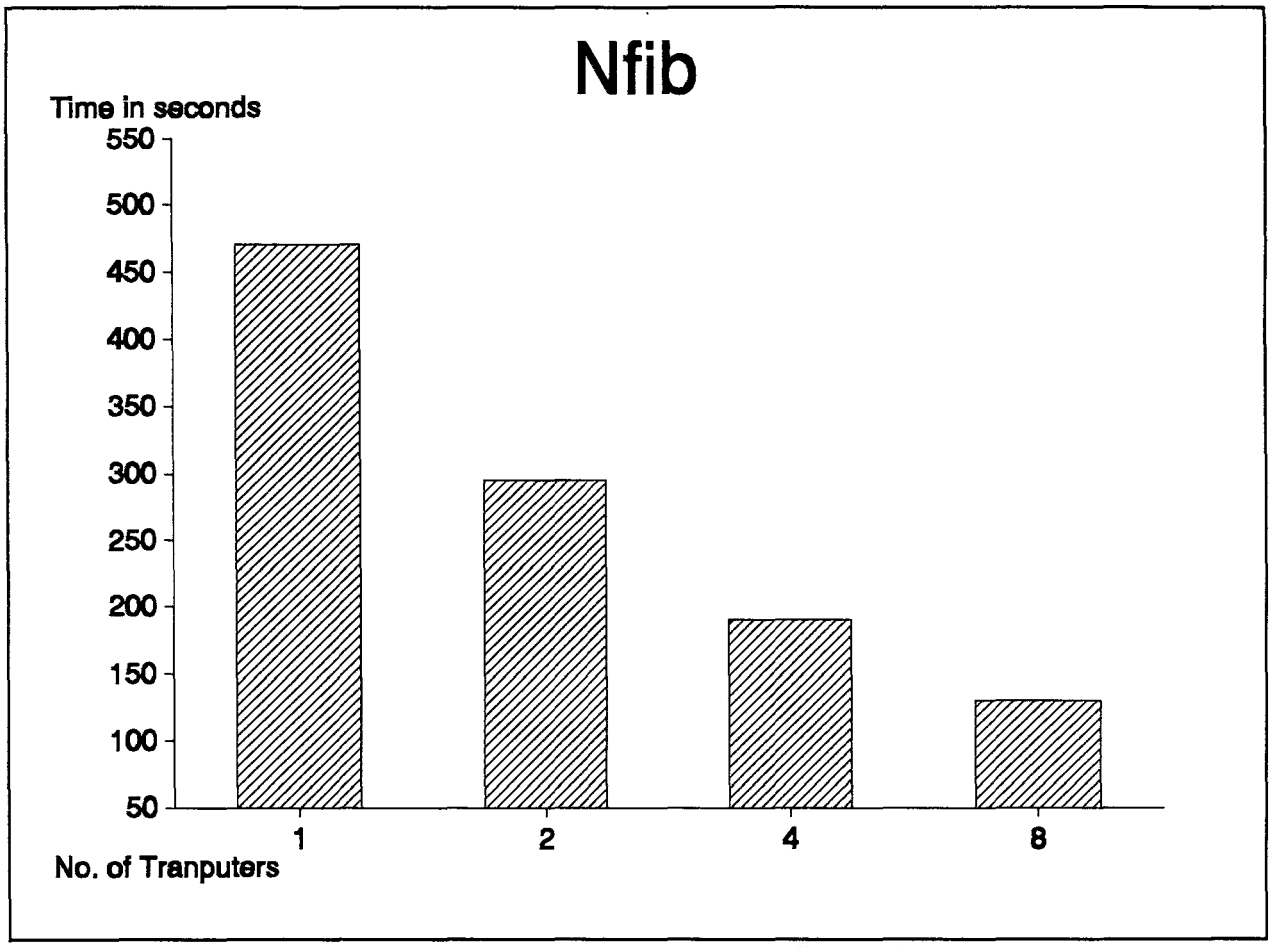

Figure 38 - Results of a skeleton application which generates the Fibonacci series.

integration problem presented (section 7.1.1) for which the communication requirements are the same.

Regular problems that result in uneven processing requirements can be tuned by altering the number of tasks each sub-tree receives. This is achieved by modifying an internal skeleton parameter. Each sub-tree maintains a queue of waiting tasks. Effectively this parameter changes the number of tasks issued to a sub-tree's queue.

7.1.3 Irregular Problems

Irregular problems are data dependant and would require significant and complex tuning to solve efficiently. Figure 39 shows a Quicksort implemented under the Divide and Conquer Skeleton. With this style of problem, the size of the sub-problems are completely unpredictable. Coupled with a large communication overhead, this produces extremely poor results. Communication costs far exceed those of computation. In the Quicksort example the size of the sub-problems, and hence the load balancing, depends on the selection of a "lucky" comparator value.

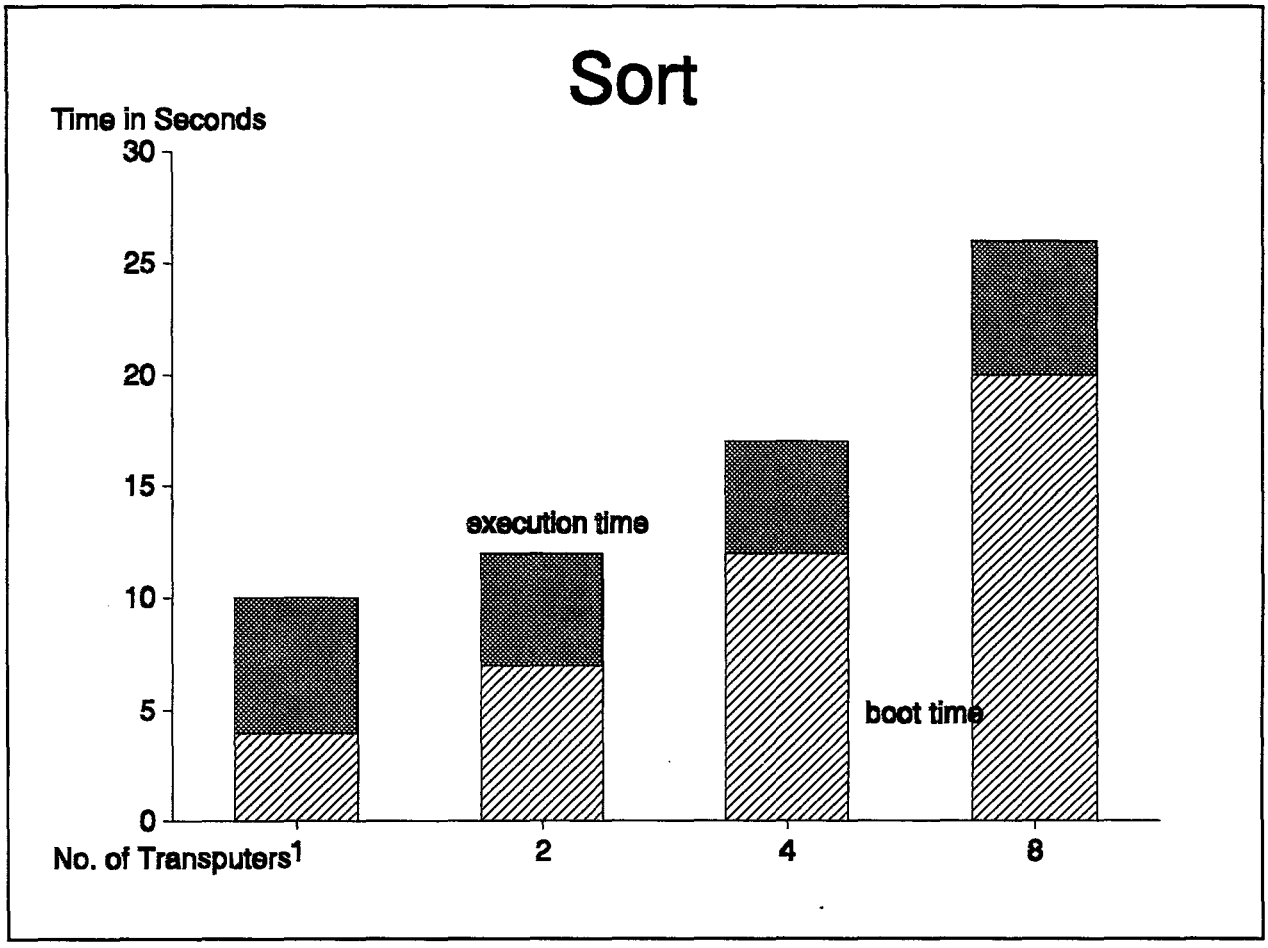

Figure 39 - Execution speed of the Sort Example.

7.2 Task Queue

The Task Queue Skeleton differs from the Divide and Conquer Skeleton in that the decompose function is a forwarding of created tasks to any processor. All nodes do computation, and there is no real combine operation. A worker returns the complete result. The Task Queue Skeleton has three features:

- There is a flow of tasks.
- Tasks are dynamically scheduled.
- Tasks are distributed from a single source.

In their paper, Feldcamp et al. [FEL91] were able to formulate queuing models to predict execution time. This is done on the basis of estimates of arrival rates of tasks, and the service rate for a task. They derive estimates of queue length, waiting times and throughput. These models don't hold when an application is communication bound. The throughput becomes limited by the rate at which tasks can be communicated, irrespective of the number of processors or the processing speeds. Feldcamp et al. developed a parameter driven method for tuning these applications to optimize performance. The parameters include:

- The number of task packets transmitted per communication.
- The maximum number of tasks assigned to a processor *i*.
- The number of processors and their interconnection.

They found that a maximum packet size was reached, after which the communication of a task could no longer overlap the computation of the previous task. Also, packaging large tasks becomes offset by the processing required to package and unpack tasks. This represents a deviation from the usual approach to skeleton research in that there is a greater interaction between the user and the skeleton.

Four virtual topologies were written for the Task Queue Skeleton, to solve each problem on 1, 2, 4 or 8 processors.

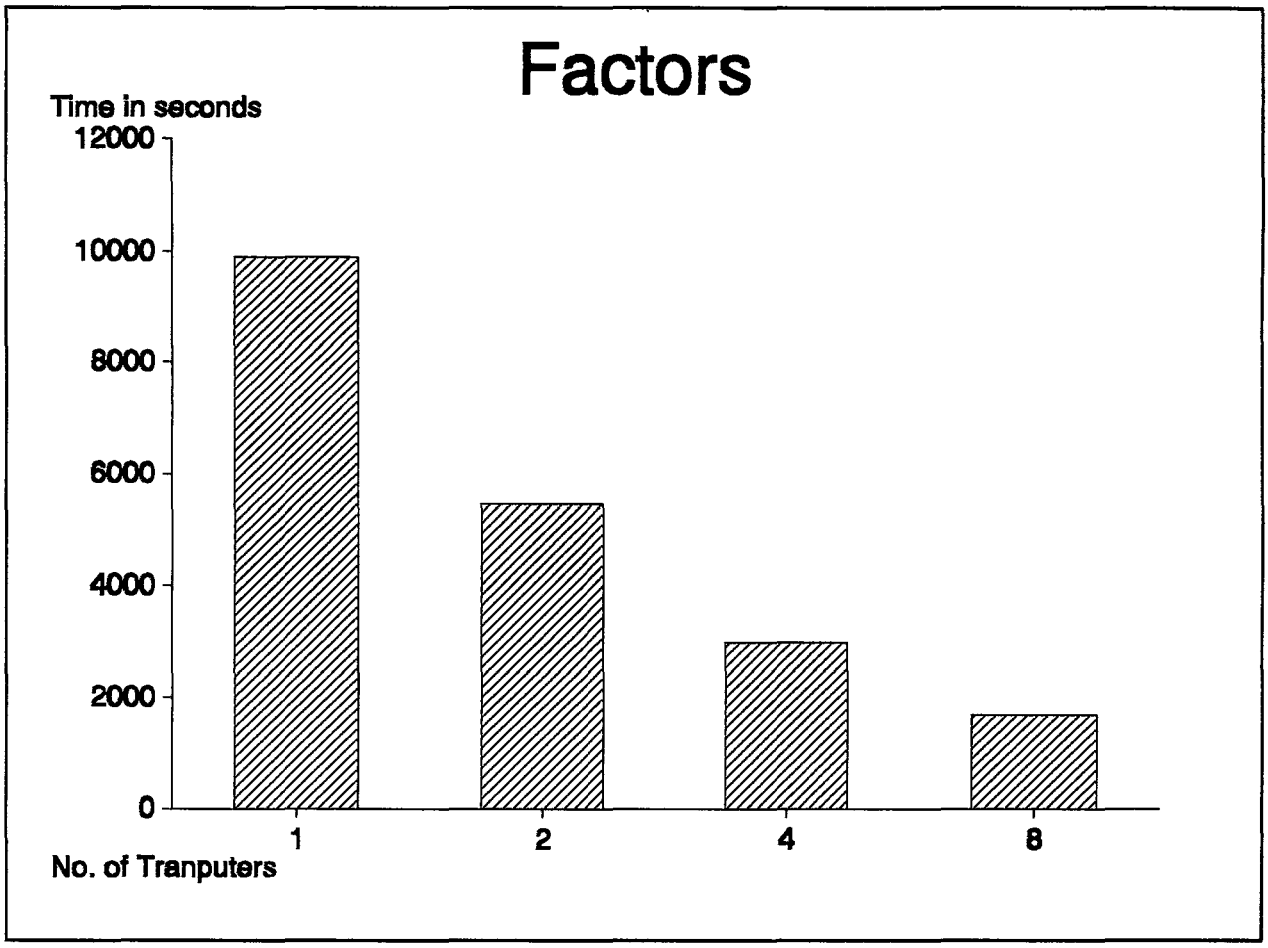

Figure 40 - Graph showing speed up of factors example.

Since load balancing occurs dynamically under a Task Queue application, the problems of Divide and Conquer applications do not apply. Communication and the swamping of resources are the most significant factors which influence performance. Two problems were implemented to show the influence communication has on performance. They are a factors example, and a solution to the Eight Queens problem (solved for a 12 by 12 board).

Figure 40 shows good scalability for the factors example. In this case the ratio of communication to computation time is low, resulting in a good improvement in execution time.

Solving the Eight Queens problems results in a large number of tasks being created. The skeleton application of Figure 41 shows a strong increase in performance between 1 and 2 processes. As

communication is significant, this improvement in performance slows as more processors are added. The creation of larger packets may improve performance, but this is a user optimization.

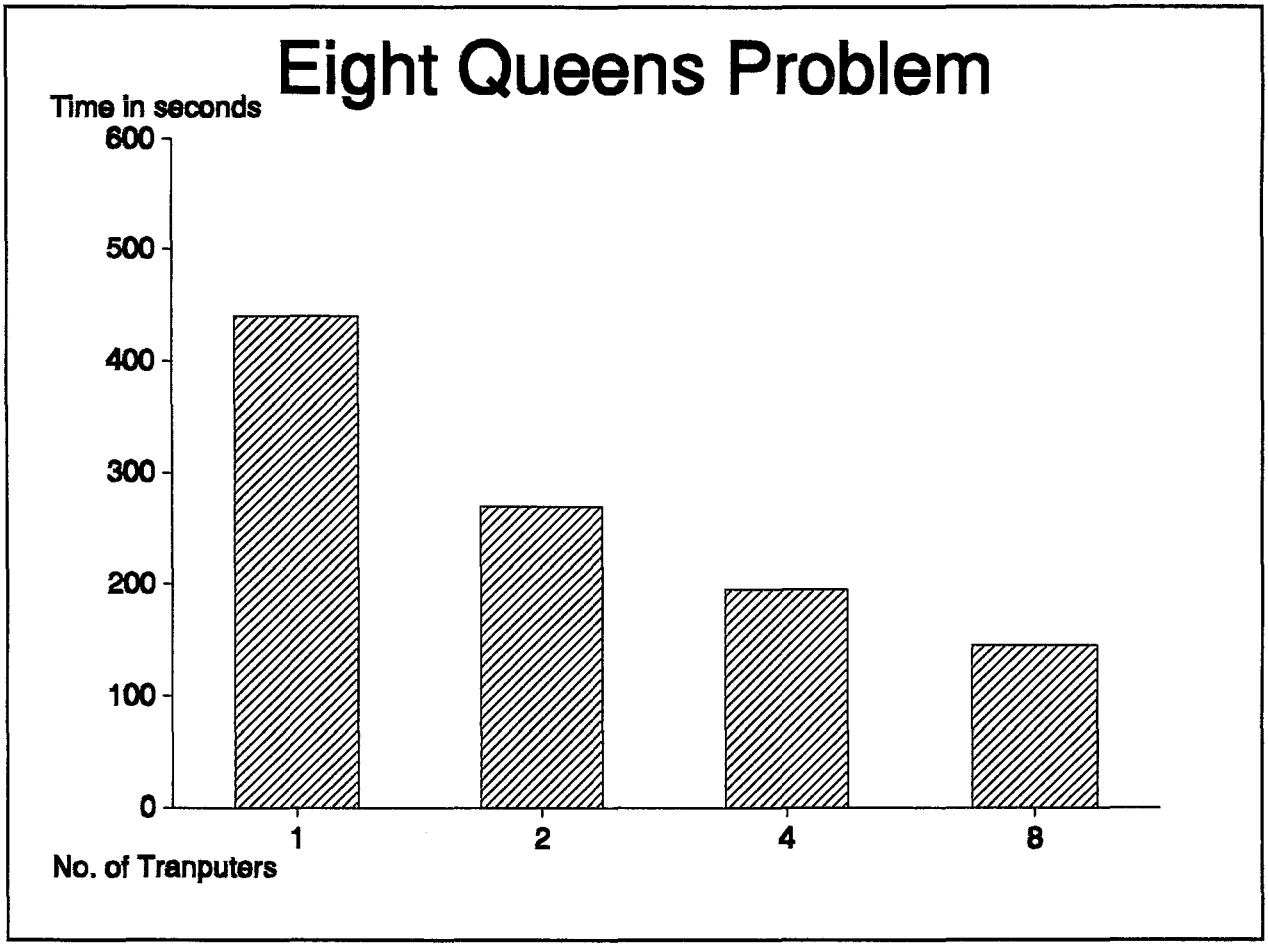

Figure 41 - Execution speed of the Queens Example.

7.3 Evaluation of Goals

A number of properties for an Algorithmic Skeleton system were identified in chapter 2. The extent to which these goals have been met is discussed below:

Hardware Independence

It was possible to divide the system into an upper and a lower level. The lower level used CDL to map the skeleton onto the hardware. This level would require reprogramming to port the skeleton. The upper

level is tied to the Helios operating system in that Helios was used to implement communication. This communication is Unix like, but Helios sets up the channels for communication automatically. In order to port the upper level changes may have to be made here. This implementation of algorithmic skeletons is configured for the MIMD type computers. Porting the skeleton implementation to other platforms would be difficult. But the application program is hardware independent.

Hidden Parallelism

The skeleton attempts to hide all parallel implementation details from the user. It was found that user intervention is required for some types of problems. Research is currently underway to increase the range of suitable problems, but all need varying degrees of user intervention[RAB90], [FEL91], [PRI88].

User Interface

Interface design is an active research field in its own right, and beyond the scope of this work. This work gave type specifications for all the user interfaces, but it did not attempt to design a elegant front-end to handle the input for the skeletons. The user had to write input procedures which were inserted into the skeleton using the 'C' preprocessor.

Independence

All skeletons were implemented independently of one another[COL89]. If more skeletons are designed it is difficult to say if independence would continue to be possible. Some research suggests that independence is not completely desirable, and that skeletons should be used to build more complex skeletons[FOS90a].

Exclusion

The skeleton should not admit applications for which it is not suitable. Once again, extensive analysis on the part of the end user, is required. The type of problem to be solved has to be identified before this best skeleton can be selected.

Performance

Performance varied, depending on the problem been solved. Good performance was shown for most problems solved under the Divide and Conquer Skeleton. Regular Balanced problems and Regular Problems scaled well, while degradation of performance resulted when Irregular problems were attempted. Good performance was shown with applications implemented under the Task Queue Skeleton.

Imperative verses Functional implementations

It was possible to implement skeletons using an imperative language. As the literature on other implementations gave few performance details, a comparison between the imperative approach of this thesis verses a functional approach was not possible.

7.4 Chapter Overview

This chapter has discussed performance issues for the Divide, and Conquer and Task Queue Skeletons. The distribution for the Divide and Conquer Skeleton suffers from three drawbacks, namely, inactive processors, exploding computational tree, and granularity problems. The extent to which this project overcame these difficulties was discussed. Divide and Conquer problems were classified as Regular Balanced, Regular, or Irregular (section 7.1). Applications were presented for each class of problem. Real improvement in efficiency was shown with Regular Balanced problems and Regular problems, and with the Task Queue approach.

The need for a user to tune a skeleton breaches the goals outlined in the introductory chapter. The extent to which this tuning is possible in this approach was discussed. Graphs were given to show the performance of the skeleton, with no tuning.

The degree to which the goals of skeleton programming were met, was discussed. It was found that the primary goal of hiding all the complexities of parallelism from the user was feasible for most test examples, but a programmer with a knowledge of parallel programming could employ the skeletons more efficiently.

Chapter 8

8. Conclusion

As parallel computers become more common, it will become increasingly necessary to find abstractions to help sequential users develop parallel programs. One approach is to attempt to hide the complexities of parallel programming from the user (see section 1.1), the most perplexing of which are listed below:

- Decomposition.
- Distribution.
- Sharing.
- Synchronization.
- Termination.

Algorithmic skeletons were proposed as a possible method of achieving this (see section 1.2). This work undertook to test this hypothesis by implementing two skeletons, Divide and Conquer (chapter 4), and Task Queue (chapter 5). In effect, it hoped to create an efficient special purpose machine for each style of parallel problem, by creating a virtual machine over the available hardware.

8.1 Evaluation of Goals - A Summary

A number of desirable qualities for algorithmic skeletons have been identified by means of a literature survey. The degree to which they were satisfied in the work of this thesis was varied. This is discussed in section 7.3, and the conclusions are summarized below.

- Hardware Independence - The system would be divided into an upper hardware independent level, and a lower hardware dependant level. This goal was largely attained.

- Hidden Parallelism - The skeleton would hide all parallel implementation details from the user. It was found that efficient implementations are possible, but that skilled user intervention can improve performance. This conclusion is corroborated in the literature [RAB90].
- User Interface - A clear and efficient user interface should exist for ease of use. This aspect was not addressed in this work. It is in itself a large area of research [GOG86].
- Independence - All skeletons should be independent of one another. This allows skeletons to be added and removed as the skeleton concept is refined. When only two skeletons are implemented, it is difficult to make any clear observations. In the implementations of this thesis, this independence was achieved.
- Exclusion - The skeleton should not allow applications for which it is not suitable. In the work of this thesis, exclusion is only achieved in that a problem is more difficult, or impossible, to implement using an unsuitable skeleton. No real success was achieved implementing this goal.
- Performance - There should be performance benefits related to the number of processors in the skeleton system. Performance was found to be dependent upon the suitability of the problem. The suitability of various divide and conquer problems and task queue is discussed below.

8.2 Divide and Conquer Skeleton

Three classes of problem were identified when using the Divide and Conquer Skeleton to develop a parallel solution (section 7.1). These problems are solved by the skeleton with varied success:

- Regular Balanced problems - The problem decomposes evenly, and the load on each processor sub-tree is equal. An example of this class is a skeleton implementation to perform integration under a curve. This problem was found to execute efficiently and to scale well.
- Regular Problems - The size of the sub-problems can be predicted, but decomposition is asymmetric, for example, an implementation to find the Fibonacci series. The test problem in this category also scaled well, but the rate of speed up tailed off as the number of processors was increased (see evaluation 7.1.2).

- Irregular problems - The size of the sub-problems cannot be predicted, and are unbalanced. An example is an implementation of a Quicksort. These problems need run time tuning and were found to be very unsuitable for parallelization with the Divide and Conquer Skeleton (see evaluation 7.1.3).

Three problems were found to exist with the established method of distribution for the divide and conquer algorithm[RAB90]. Under this method, the Divide and Conquer Skeleton's distribution is that of a tree of processors.

- The processors executing the interior nodes of the tree are inactive for most of the time.
- The computational tree is much larger than the physical machine.
- The fine granularity causes excessive communication overheads.

The distribution presented in this thesis solves the first two shortcomings (see section 4.4). However the approach suffers from too fine a granularity when solving problems requiring large amounts of data.

8.3 Task Queue Skeleton

There are two major problems associated with implementing Task Queue Skeletons on a physical multi-processor architecture[RAB90]:

- Tasks which are too tiny in comparison to the communication overheads they generate, execute faster sequentially.
- If too many tasks are generated by a problem, the systems resources are overwhelmed.

The dynamic load balancing of the Task Queue Skeleton resulted in both example applications scaling well. The factors example showed good speed-up as the number of processors increased. In the case of the Eight Queens example there was a tailing off of performance. This was caused by the large number

of tasks created, and the resulting communication overheads. The creation of larger packets of data for communication may improve performance, but this is a user optimization.

The need to tune skeleton applications breaches the goals outlined in the introduction. Graphs were presented in chapter 7 to show the performance of the skeleton with no tuning.

8.4 Future Research

Current implementation techniques for skeletons have been criticized for their lack of flexibility[FOS90a]. In these approaches, it is difficult to find the optimum skeleton for a user problem because applications must be written for a specific skeleton. The mechanisms by which greater flexibility is possible (Chapter 6) are those of "modification" and "composition". Modification allows a user application to be tested with different skeleton implementations, whilst composition allows the features of a skeleton to be combined with those of another to form a new skeleton.

8.5 Final Observations

It was found that the primary goal of hiding all parallelism from the user was feasible, but that a programmer may need knowledge of parallel programming to use a skeleton more efficiently. Problems suited to fine grained parallelism scale particularly well, but other problems may require detailed analysis and dynamic load balancing to improve efficiency further. Heavy communication requirements will cause degradation in performance, a common feature of MIMD architectures.

Implementing skeletons efficiently is not as easy as it initially appears. The idea of moving the parallel complexity from the application programmer to the implementor of the skeleton doesn't immediately solve the complexity problem, it merely moves the responsibility to the implementor. Here the problems are more complex than before, as the implementor has to provide an efficient general solution. Initial evidence from this study shows that this may be possible, but that extra knowledge or hints about the user's specific problem may sometimes be required.

Appendix A - Introduction to CDL

The purpose of CDL (Component Distribution Language) is to provide a high level approach to the physical distribution of code and the establishment of communication channels between processes.

Once program components have been written and their interconnections have been described using CDL, the actual physical distribution and establishment of channels can be done by CDL. The proximity of processes is not a concern to the user as all messages are routed by CDL's Task Force Manager.

CDL simplifies parallel programming for people already familiar with the concept. The user must still describe the code distribution and communication channels. It does not solve problems such as decomposition, it only facilitates implementation. Using CDL, the actual coding of the implementation is simplified.

In CDL, each *component* of user code (tasks) can be given specific resources using "Component Declarations":

- Code - The code that constitutes a particular Task.
- Processor - The type of transputer required T414, T870 or Any.
- Puid - The physical processor to execute a Task.
- Attrib - May be used to assign attributes to certain processors, and then specific tasks can be assigned to processors with specific attributes.
- Memory - How much is required by a task?
- Streams - Streams are used to explicitly specify a stream on which a process must communicate.

The above fields need not necessarily be defined for user tasks. There are default settings, which may or may not be the most efficient for a particular problem.

CDL creates Unix style communication channels between components. Using CDL the interconnectivity is specified as follows:

A | B	Communication channel from A to B only.
A <> B	Channel from A to B and from B to A.
A ^^ B	A and B execute independently in parallel, no channel.
A ||| B	A load balancer is inserted between A and B.
A (Auxiliary List)	A has additional communication channels apart from the one to B.

CDL Load Balancer

CDL provides a load balancer for "farm of worker" type applications. A master process writes tasks to the load balancer, which buffers them and issues them to idle workers.

The load balancer forks off a process for each of the workers, and communication is established between this process and the worker. The forked process reads a job from the load balancer's buffer, and writes it to its associated worker. The worker then executes this job. Before a new task is issued a solution must be passed to the master via the workers associated communication process in the load balancer.

Modification to the Load Balancer

The original load balancer expected exactly one solution from a worker. In order to allow the worker to pass back any number of solutions the following modifications were made. Two processes, rather than one, were forked for each worker. One issues jobs to its associated worker, and the other reads solutions from the worker. This allows any number of solutions to be written back to the load balancer. The worker sends an end-of-job message to allow the master to keep track of outstanding jobs. The alteration to the load balancer is transparent to existing applications.

Bibliography

[AHO74] Aho A., Hopcroft J., Ullman J., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[ALA78] Alagic S., Arbib M., *The Design of Well Structured and Correct Algorithms*, Springer-Verlag, Berlin, 1978.

[ALE86] Alexandridis N., Adaptable Software and Hardware: Problems and Solutions, *IEEE Transactions on Computers*, February 1986.

[BOI87] Boillat J., Kropf P., Meier D., Wespi A., *An analysis and Reconfiguration Tool for Mapping Parallel Programs onto Transputer Networks*, Institute of Informatics, University of Berne, 1987.

[BRO89] Browne J., Werth J., Lee T., *Intersection of Parallel Structured and Reuse of Software Components: A Calculus of Composition for Parallel Programs*, International Conference on Parallel Processing, 1989.

[COL89] Cole M., *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge, 1989.

[CHA88] Chandy K., Misra J., *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[FEL91] Feldcamp D., Sreekantaswamy H., Wagner A., Chanson S., Towards a Skeleton-based Parallel Programming Environment, in *Transputer Research and Applications 5*, editor Veronis A., Paker Y., IOS Press, 1991.

[FOS89] Foster I., Taylor S., Strand: New Concepts in Parallel Programming, Prentice-Hall, 1989.

[FOS91] Foster I., Automatic Generation of Self-Scheduling Programs, *IEEE Transactions on Parallel and Distributed Systems*, vol. 2 No. 1, January 1991.

[FOS] Foster I., *On the Refinement and Composition of Process Structures*, technical document IL60439, Argonne National Laboratory.

[FOS90a] Foster I., Stevens R., *Parallel Programming with Algorithmic Motifs*, IL60439, Argonne National Laboratory, 1990.

[FOS90b] Foster I., Kesselman C., Taylor S., *Concurrency: Simple Concepts and Powerful Tools*, Mathematics and Computer Science Division, Argonne National Laboratory, 1990.

[GIN87] Ginn J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[GOG86] Goguen J., Reusing and Interconnecting Software Components, *IEEE Transactions on Computers*, February 1986.

[HAR87] Harel D., *Algorithmics: The Spirit of Computing*, Addison-Wesley, 1987.

[HEL91] *The Helios Parallel Operating System*, Prentice-Hall, 1991

[HOR83] Horowitz E., Zorat A., Divide and Conquer for Parallel Computing, *IEEE Transactions on Computers*, vol. C-32, pp. 582-585, June 1983.

[HOR73] Horowitz E., Sahni S., *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[JAM88] Jamieson L.H., Gannon D.B., Douglass R.J., *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, 1988.

[KAC90] Kacsuk P., *Execution Models of Prolog for Parallel Computers*, MIT Press, 1990.

[KEL87] Kelly P., *Functional Programming for Loosely-coupled Multiprocessors*, MIT Press, Cambridge, 1989.

[QUI87] Quinn J., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[RAB90] Rabhi F., Manson G., Divide and Conquer Parallel Graph Reduction, *Parallel Computing*, vol. 17, pp. 189 to 205, 1990.

[SED88] Sedgewick R., *Algorithms*, Addison-Wesley, 1988.

[SHI90] Shizgal I., *The Amoeba-Prolog System*, The Computer Journal, vol. 33 No. 6, 1990.

[SIN91] Singh A., Schaeffe J., Green M., A Template-Based Approach to the Generation of Distributed Applications Using a Network of Workstations ,*IEEE Transactions on Parallel and Distributed Systems*, Vol 2, No 1, January 1991.

[THO84] Thomas, J.R., Finney R.L., *Calculus and Analytic Geometry*, Addison-Wesley, 1984.

[TRA89] *Transputer Technical Notes*, Prentice Hall, 1989.

[WAG92] Wagner A., Personal communication, University of British Columbia, 1992.