On the implementation of P-RAM algorithms on feasible SIMD computers

A thesis submitted

for the degree of Doctor of Philosophy

by

RIDHA ZIANI

Department of Computer Science

University of Warwick

August 1992

ACKNOWLEDGMENTS

I wish to express my gratitude to Dr. Alan Gibbons for his supervision, guidance and patience during the years I spent at Warwick.

I also would like to thank the members of the Computer Science Department for all the assistance I had and the Algerian government for financial support.

Finally, I am grateful to my family, to Janette and to many of my friends for all the moral support I had during the years.

DECLARATION

This is a thesis submitted to the University of Warwick in support of my application for admission to the degree of Doctor of Philosophy. It contains the account of my own work performed in the Department of Computer Science of the University of Warwick under the general supervision of Dr. Alan Gibbons. No part of it has been submitted in support of an application for another degree or qualification of this or any other institution of learning. The work described in this thesis is the result of my own independent research except where specifically acknowledged in the text.

Parts of this work have been presented or appeared as follows:

(i) *Techniques for the efficient implementation of some P-RAM algorithms on the Mesh-connected computer*, 5th British Colloquium on Theoretical Computer Science, Royal Holloway and Bedford New College, London University, Egham, April 1989.

(ii) *The Balanced binary tree technique on Mesh-connected computers*, Information Processing letters 37, January 1991, 101-109.

On the implementation of P-RAM algorithms on feasible SIMD computers

ABSTRACT

The P-RAM model of computation has proved to be a very useful theoretical model for exploiting and extracting inherent parallelism in problems and thus for designing parallel algorithms. Therefore, it becomes very important to examine whether results obtained for such a model can be translated onto machines considered to be more realistic in the face of current technological constraints.

In this thesis, we show how the implementation of many techniques and algorithms designed for the P-RAM can be achieved on the feasible SIMD class of computers.

The first investigation concerns classes of problems solvable on the P-RAM model using the recursive techniques of compression, tree contraction and 'divide and conquer'. For such problems, specific methods are emphasised to achieve efficient implementations on some *SIMD* architectures. Problems such as list ranking, polynomial and expression evaluation are shown to have efficient solutions on the 2−dimensional mesh-connected computer.

The balanced binary tree technique is widely employed to solve many problems in the P-RAM model. By proposing an implicit embedding of the binary tree of size n on a $(\sqrt{n} \times \sqrt{n})$ mesh-connected computer (contrary to using the usual H-tree approach which requires a mesh of size $\approx (2\sqrt{n} \times 2\sqrt{n})$), we show that many of the problems solvable using this technique can be efficiently implementable on this architecture. Two efficient $O(\sqrt{n})$ algorithms for solving the bracket matching problem are presented. Consequently, the problems of expression evaluation (where the expression is given in an array form), evaluating algebraic expressions with a carrier of constant bounded size and parsing expressions of both bracket and input driven languages are all shown to have efficient solutions on the 2−dimensional mesh-connected computer.

Dealing with non-tree structured computations we show that the Eulerian tour problem for a given graph with m edges and maximum vertex degree d can be solved in $O(d\sqrt{m})$ parallel time on the 2−dimensional mesh-connected computer.

A way to increase the processor utilisation on the 2-dimensional mesh-connected computer is also presented. The method suggested consists of pipelining sets of iteratively solvable problems each of which at each step of its execution uses only a fraction of available PE's.

The techniques and subproblems investigated in this thesis are of such commonality in the design of parallel algorithms that they could be usefully implemented as a library of resources on feasible machines.

Contents

Chapter 1

Designing algorithms for parallel computers

1.1 Introduction

Unlike serial computation where a more unified approach is taken in the design and analysis of algorithms, the situation in parallel computing is quite different. To say the least, the variety of parallel algorithms that exist, do not all fit in a general framework (see, e.g. [GR88], [A85], [A89], [U84]) since the intricacies brought to light by the idea of making a collection of processors cooperate to achieve a task, are not fully understood.

Amongst many issues the notions of organisation and nature of the parallel models of computation have been a dividing factor in the community of algorithmic researchers. Those who are motivated by mere intellectual challenge have taken many liberties regarding the feasibility factor of parallel machines. In contrast, those who have been motivated by the desire of making full use of current available parallel computers are being more realistic regarding the technological limits. In this respect, parallel machines can be separated into two broad categories, namely the abstract or ideal models and the more realistic or feasible ones.

Parallel algorithms differ from their sequential counterparts in the approach used in their design. The different ways for proceeding to solve a problem on a parallel machine are to parallelise a sequential algorithm, to adapt a parallel solution from one machine to another or simply to design a new solution right from scratch.

Benefiting from previous work by trying to detect and exploit any inherent parallelism in an existing sequential algorithm is a task that has proved not to be easy. Many elegant sequential solutions to some problems have been found very hard to parallelise, they include problems such as depth-first search [V91], the Eulerian circuit problem [AIS84] or simply the very old problem of finding the great common divisor of two natural numbers [H87].

The second alternative which is to implement or readapt a parallel algorithm initially designed to run on a different model is very appealing. However, practice has shown that it has many drawbacks if some issues such as inter-processor communications are not properly handled.

When there is no possibility to follow the above paths then the last resort is to invent a new parallel algorithm right from scratch.

This thesis follows the second approach. It will be shown that many algorithms designed for an ideal model such as the P-RAM (using a predefined amount of time and resources) can be implemented on more feasible machines within realistic bounds. This is mainly achieved as follows:

1) by adapting the algorithmic techniques used in the design of these P-RAM algorithms.

2) by implementing some basic and widely used P-RAM tools incorporated in these algorithms.

The algorithmic techniques treated in this thesis are introduced in chapter 2 along with techniques of a different nature which also share the goal of facilitating the design of parallel algorithms. In chapter 3 we present a set of tools or library candidate routines which include widely used simple P-RAM algorithms and utilities as well as routines that are essential to use in a realistic parallel setting.

But prior to this, the rest of this chapter is devoted to the different models of parallel computation concentrating mainly on the SIMD class of computers. The differences that exist amongst them as well as the ways in which they relate to each other are highlighted. It will also present the limits of these parallel models by introducing a few notions from parallel complexity theory.

1.2 Machine Models of parallel computation

In the field of parallel algorithmic design, researchers and problem solvers have used several models of computation, ranging models such as the sorting network of the three Hungarians Ajtai, Komlos and Szemerdi [AKS83] to the various machines that are now commercially available like the highly publicised *Connection Machine* [Hi85]. The differences between these lie essentially in their structure on the one hand and on their behaviour or the way in which they handle data on the other. As it will be seen later, these factors allow a clear distinction to be made between what is by today's technological standards a theoretical model in contrast to what is called a practical or realistic model. This abundance of models has pushed for their categorisation or classification under different criteria. In the literature many different such classifications exist. We have retained the two most commonly used ones and from which the terminology of this thesis is borrowed. They are the classifications of Flynn [F66] and Schwartz [S80].

1.2.1 Flynn's classification

The earliest and most used classification of parallel models of computation, based on the notion of synchronicity and the number of data and instruction streams handled in parallel, is due to Flynn [F66] who distinguishes four classes of machine as follows:

1. **SISD** (Single Instruction stream, Single Data stream) class. Machines in this class are those performing one instruction at a time on one set of data.

The traditional sequential computers belong to this class.

2. **SIMD** (Single Instruction stream, Multiple Data stream) class. This class contains the parallel machines that allow the simultaneous execution of one instruction on possibly different sets of data. A so-called *enable/disable* mask (e.g. an *if - then* block) selects the processing elements that are allowed to execute operations on their assigned data. The ICL/DAP (Distributed Array Processor), The ILLIAC IV, the Burroughs PEPE and the Goodyear Aerospace MPP are examples of computers that belong to this class [HB85].

3. **MISD** (Multiple Instruction stream, Single Data stream). This category which has received very little attention except in domains such as signal and image processing (computer vision) comprises machines that perform multiple sets of instructions on a single stream of data [A89].

4. **MIMD** (Multiple Instruction stream, Multiple Data stream) class. This class, which is considered as the most general and most powerful class of machines, includes those performing different sets of instructions on different sets of data. An MIMD computer is either synchronous or asynchronous. In the former case all processing elements perform each successive set of instructions simultaneously, whereas in the latter, the processors run independently and wait only if information from other processors is needed. An example of an asynchronous MIMD machine is the Denelcor/HEP (Heterogeneous Element Processor) [HB85].

1.2.2 Schwartz's classification

Schwartz [S80] classifies parallel computers according to the method in which information is passed amongst the processors; an issue which is of crucial importance in parallel computing environments. He calls a *paracomputer*, a parallel computer whose processors can have simultaneous access to a shared common memory and thus who can communicate in constant time. Whereas a parallel computer where each processor has its own memory and where inter-processor communication is achieved only via a fixed interconnection network is referred to as an *ultracomputer*.

1.2.3 The SIMD class of computers

The focus of this thesis is on the SIMD class which consists of the two categories of paracomputers and ultracomputers where PE's operate synchronously in a lock-step fashion. That is, the PE's are synchronised to perform the the same function at the same time . The next sections look at the standard paracomputer or shared memory model of computation (the P-RAM) as well as widely used interconnections networks that have characterised ultracomputers in general.

1.3.2.2 The P-RAM model of computation

The most popular model amongst parallel algorithms designers is the P-RAM (Parallel Random-Access Machine) model introduced by Fortune and Willie [F78]. This model is much liked because of its simplicity and great power to express

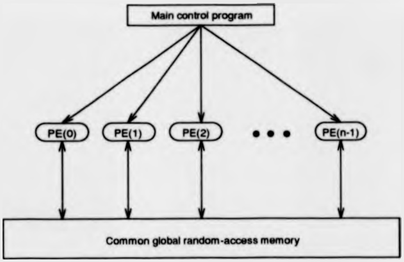


Figure 1.1: The P-RAM diagram

parallelism. A P-RAM is a collection of n processors (throughout the thesis, the abbreviation PE will be used to denote a processing element and PE(i) will denote the processor with index i) indexed from 0 to $n-1$ which synchronously execute the same program (through the central main control) and which communicate via a common random access global memory. Each processor is a RAM (Random Access Machine [AHU74]) capable of executing standard operations in constant time. Figure 1.1 schematically shows the P-RAM model.

The P-RAM model neglects the hardware limitations that an actual parallel computer would impose, particularly arising from how the processors are connected. It implicitly assumes that all different connections between processors and memory locations exist and thus communication takes constant time. Such an assumption is unrealistic due to the impracticality of wiring processor to memory when the number of processors and the size of the memory are large. However this does not alter the fact that the P-RAM is a very powerful model for the design of parallel algorithms in general and explicitly employs the parallelism of problems.

Variations of the P-RAM model exist, which are based on protocols for reading and writing information from the global memory. Whether a model will allow or prohibit concurrent reads (many PE's trying to read the same memory location) or concurrent writes (many PE's trying to modify the contents of the same memory location), affects its strength. Consequently three subclasses of the P-RAM model are distinguished in order of increasing strength:

(i) Exclusive-Read, Exclusive-Write (EREW) P-RAM. In this model no two processors are simultaneously allowed to read from or write into the same memory location.

(ii) Concurrent-Read, Exclusive Write (CREW) P-RAM. Many processors are allowed to synchronously read from the same memory location, but no concurrent writes are allowed.

(iii) Concurrent-Read, Concurrent-Write (CRCW) P-RAM. Both multiple-read and multiple-write are allowed.

Writing conflicts of the CRCW are resolved by setting arbitration rules among contending processors. Some commonly employed resolution methods are:

(a) All processors writing into the same memory location must write the same value. If such a rule is adopted, then the model is called a *Common* model.

(b) Any processor involved in a writing conflict may succeed and a task performed by the model must work correctly regardless of which one succeeds. A model where this rule is chosen is called an *Arbitrary* model.

(c) The minimum indexed processor in a conflict. If this rule is carried out then the model is called a *Priority* model.

The following [Ha91] illustrates the relative strengths of the variations of the P-RAM model:

$$EREW \leq CREW \leq Common \leq Arbitrary \leq Priority$$

1.3.2.2 Feasible SIMD models

By a feasible or practical model of computation, it is meant (in contrast to the P-RAM model) a machine that can be constructed using current technology.

A feasible SIMD computer (as shown in figure 1.2) consists of n processing elements (indexed from 0 to $n-1$) all of which are under the control of on control unit (CU) and communicating through an interconnection network. Each PE has its own working registers and memory (M). The CU has also its own memory for the storage of programs which can be loaded from an external source. The function of the CU is to determine where the instructions should be executed and subsequently bradcasted to the appropriate PE's. All the PE's perform the same function synchronously in a lok-step step fashion under the command of the CU. Data is loaded into the PE's from external sources via a data bus or via the CU.

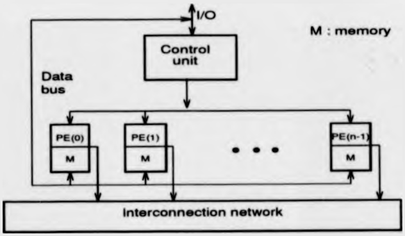


Figure 1.2: The diagram of a feasible SIMD computer

PE's may be active or disabled for executing a given operation [HB85]. Data exchanges among the PE's are achieved via the interconnection network which is also under the control of the CU.

Feasibility of machine models of parallel computation also depends on the number of links from each PE in the network being bounded by a manageable integer (i.e. constant or at most growing logarithmically with the size of the network) and the maximum path length within an architecture being small enough to allow fast communications. These two quantities are often referred to as the degree (d) and diameter (D) of the architecture. A simple block diagram for an SIMD computer is shown in figure 1.2.

Different network topologies lead to different SIMD architectures in respect to the parameters diameter and degree. In what follows we present a catalogue of networks referring occasionally to PE's as nodes of a graph (the interconnection network).

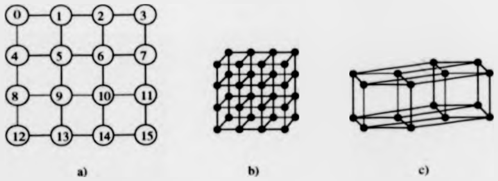


Figure 1.3: 2, 3 and 4-dimesional meshes

a) The Mesh-Connected Computer (*MCC*) family. The mesh-connected class of computers has received wide attention in the literature merely because of the fact that one of the first commercialised parallel computers, the ILLIAC IV had a mesh structure. In general such a structure may be thought of as a collection of n PE's logically arranged in a q-dimensional array $A(n_{q-1}, n_{q-2}, \ldots, n_0)$, where n_i is the number of PE's in the i^{th} dimension and $n = n_{q-1} \times n_{q-2} \times \ldots \times n_0$. The PE at location $A(i_{q-1}, \ldots i_0)$ is connected to the PE's at location $A(i_{q-1}, \ldots i_{j\pm1}, \ldots, i_0)$, $0 \leq j < q$, provided they exist. Varying the sizes of the dimensions of a mesh obviously modifies the diameter (D) of the architecture which is given by the simple formulae $D = \sum_{q=1}^{m}(n_{q-1} - 1)$ and the degree d which is bounded by $2q$. When all the dimensions have the same size, $D = q(n^{1/q} - 1)$. Figure 1.3.(a) shows a (4×4) *MCC* where the processors are indexed according to a natural order i.e. from left to right, top to bottom. Other indexing schemes for the 2-dimensional *MCC* (MCC^q will represent a q-dimensional mesh) will be seen in later sections. Figures 1.3.b and 1.3.c show respectively $(4 \times 4 \times 2)$ and $(2 \times 2 \times 2 \times 2)$ meshes.

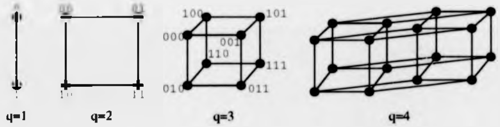


Figure 1.4: 1, 2 3 and 4-dimensional hypercubes

b) The Cube-Connected Computer (CCC) or hypercube family. The hypercube topology is one of the most common structures adopted for many recent parallel computers such as (for instance) the famous Connection Machine mentioned earlier [Hi85], [TW91]. A hypercube or CCC of dimension q is a machine with $N = 2^q$ PE's each having a distinct label or index $i \in \{0..N-1\}$ such that links exist only between the PE's having the binary representation of their indices differ in exactly one position. Formally PE(i) with the binary representation of i being $i_{q-1} \ldots i_0$, is connected to the q PE's with the binary indices equal to $i_{q-1} \ldots c(i_b) \ldots i_0$, where $c(i_b)$ is the complement of i_b and $0 \leq b < q$. The degree and diameter of the hypercube architecture are both equal to $\log N = q$ (all logarithms used in thesis are of base 2). The recursive construction of a hypercube of dimension q from 2 hypercubes of dimension $q-1$ is shown in figure 1.4 for the hypercubes of dimensions 2, 3 and 4.

c) The perfect shuffle and the deBruijn family of networks. Another architecture having the desired property of small degree and diameter is the perfect shuffle network. Like the hypercube, the perfect shuffle computer (PSC) or shuffle exchange network of dimension q has $N = 2^q$ PE's. Each processor whose index is

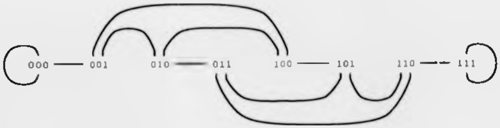


Figure 1.5: A shuffle-exchange network with 8 processors

connected to the three processors PE(j), PE($shuffle(i)$) and PE($unshuffle(i)$). The operations $shuffle$ (cyclic left shift) and $unshuffle$ (cyclic right shift) are defined on i as follows : if the binary representation of i is $i_{p-1}i_{p-2}\ldots i_1i_0$, then $shuffle(i) = i_{p-2}\ldots i_1i_0i_{p-1}$ and $unshuffle(i) = i_0i_{p-1}\ldots i_1$. PE(j) denotes the PE whose index j differs from i only in the least significant bit. The connections PE(i) to PE(j) are called *exchange* connections and the remaining are referred to as *shuffle* connections. A PSC has constant vertex degree $d = 3$ and its diameter is $2\log N - 1$. Figure 1.5. shows a shuffle-exchange network with $q = 3$.

A deBruijn network is similar to the shuffle-exchange network except that the exchange connections are replaced by the so-called 'exchange-shuffle' connections. This last type of connection links PE(i) to PE(j) if and only if the binary representation of i and $shuffle(j)$ differ in their least significant bit [L92]. A deBruijn network has $d = 4$ and $D = \log N$ as shown in figure 1.6 for such a network with $q = 3$.

d) The Tree Structured Computer (TSC) family. Trees are an important tool in structuring computations in both the sequential and parallel domains and

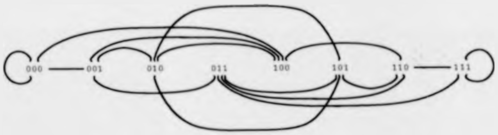


Figure 1.6: A deBruijn network with 8 processors

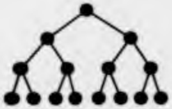


Figure 1.7: A tree network with 15 processors

therefore it is natural that they have been proposed as useful networks, at least for some applications. A tree structured network is a collection of $N = 2^p - 1$ PE's forming a complete binary tree with p levels numbered 0 to $p-1$. The PE at level i is connected to its parent at level $i+1$ (except when it is the root) and to its two children at level $i-1$ (except when it is a leaf). The degree of the tree architecture is obviously bounded by 3 and its diameter is $2\log(N+1)$. A (15 PE) TCC is illustrated in figure 1.7.

e) Other networks. Other networks commonly encountered are considered to be variations or enhanced versions of some of the architectures mentioned above. For instance the cube connected cycles computer and the butterfly network can be regarded as hypercubes with a fixed number of connections for every PE. The cube connected cycles computer is a hypercube where each of the 2^q processors

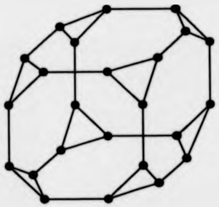


Figure 1.8: A cube-connected cycles network with 24 processors

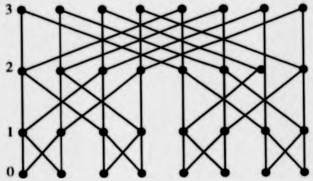


Figure 1.9: A butterfly network with 32 processors

is replaced by a cycle of q PE's, hence it has $n = q2^q$. The diameter of the architecture is $2 \log n$ and the degree is 3. Figure 1.8 illustrates such an architecture for $q = 3$.

The butterfly network of dimension q has $n = (q + 1)2^q$ processors organised in $(q + 1)$ ranks numbered from 0 to q and each having 2^q PE's.The diameter of the architecture is $2 \log n$ and the maximum degree is 4. Figure 1.9 shows a butterfly of dimension 3.

Finally, other networks in the same vein are the X-tree computer shown in

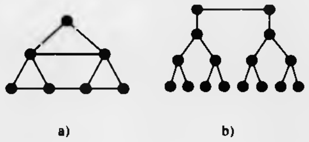


Figure 1.10: The X-tree and double rooted binary tree networks

figure 1.10 (a) with 3 levels, and the double rooted binary tree structured computer shown (with 4 levels) in figure 1.10 (b) which are regarded to be variations of the Tree Structured Computer. Some additional networks appear under the headings of future sections.

For the sake of comprehending the limits of parallel models of computation in general, we review a few notions in complexity theory.

1.3 Complexity theory of parallel computation

Bearing in mind the impact that complexity theory has had in sequential computation, a similar theory in the parallel field has proved essential. This section outlines aspects of that theory relevant to algorithmic design which lead to an understanding of the theoretical limits of parallel models of computation as well as providing important tools for assessing parallel computation itself.

1.3.1 Limits of parallel models of computation

Limits for parallel computation can be derived on different bases. On an upper level, these limits are in strict relation with the nature of the problems to be solved. That is, problems could be classified in terms of computations, between two extremes. On the one hand, there are those whose computations could be entirely fragmented into independent tasks and thus are well suited to be executed in parallel constant time. On the other hand, there are those that are basically sequential in nature and remain hardly parallelisable even if unbounded parallel execution is available.

Regarding intractability, the question whether parallelism can be used to solve intractable (*NP* hard) problems in reasonable (polynomial) parallel time is pertinent. On sequential models of computation, *NP* hard problems have polynomial time solutions if nondeterminism is used [Ka86]. Emulating the strategy of guessing by using sufficient numbers of PE's on parallel machines, will also lead to what seems to be reasonable parallel time solutions even for the NP-complete problems. Nevertheless if that seems to indicate that intractability is removed by using parallelism, we note that in order to obtain such solutions, the number of PE's needed becomes impractical since computation trees will have exponential path lengths.

For problems that are conjectured to have no reasonable sequential solutions, the question whether these problems have reasonable parallel solutions is still open. This is shown by a key concept in parallel complexity theory which is the so-called parallel computation thesis, see, e.g ([G82], [CKS81]). This thesis states that '*time bounded parallel machines are polynomially related to space bounded sequential machines*'. In other words this implies that if a problem is solved sequentially using a certain amount of space, say $S(n)$ for inputs of length n, then it can be solved in parallel in time that is no worse than $S(n)^{O(1)}$ (i.e. polynomial in $S(n)$), this is symbolically written : *Sequential-PSPACE = Parallel-PTIME*. Thus our question is simply reduced to whether the class PSPACE contains problems that can be proved not to have polynomial time sequential solutions.

Other limits that can be derived are those considering technological constraints on certain models of computation. Using Schwartz's terminology [S80], it is an obvious fact that an *ultracomputer* can never solve a problem faster than a *paracomputer* due to communication overheads. However , an *ultracomputer* could match the performance of a *paracomputer* in terms of time complexity for the solution of a given problem, if the communication pattern is good enough. As a consequence any theoretical upper limits derived for ultracomputers hold for paracomputers and conversely for the lower limits.

On an ultracomputer the performance of an algorithm may depend on the relationship between the quantity (n) of data handled and the number (p) of PE's available (This problem is often ignored on paracomputers because of the allowance of unbounded parallelism). In real applications one could be faced with the problem of having fewer PE's than data items $(n > p)$.

In contrast there is the rare situation where one has to solve a problem with fewer data items than the PE's available. In this case the processors that could have been idle may be used to speed up calculations. For instance Nassimi and Sahni have shown (see [S80]) that $n^{1-\delta}$ numbers can be sorted in $O(\delta^{-1}\log n)$ time on an n-processor ultracomputer despite the fact that n elements are sorted in $O(\log^2 n)$ time.

For some problems the first situation (as it will be seen in more detail) can be adopted to achieve optimal speedups. For example, to compute the sum or product of n integers, a straightforward way could be to structure the computation in a a binary tree form by assigning one integer to every PE at the lowest level of the tree. Using such mappings these type of computations can be achieved in $O(\log n)$ due to inter-processor communication as we go up along the tree. This blind version of a sequential linear time algorithm is considered as not fast enough and can be much improved by distributing data more efficiently.

1.3.2 The NC class of problems

The major reason for introducing parallelism in problem solving on computers is unarguably to reduce running times. It is in accordance with this exigence that the class *Parallel-PTIME* (the class of problems solvable in parallel polynomial time) cannot really be considered as a good standard for specifying efficient solutions.

Consequently, another norm to characterize efficiency in parallel environments was put forward. This norm conveys a main objective namely the achievement of sublinear running times for solving a wide range of problems. Solutions are regarded as efficient only if they are achieved in polylogarithmic time and using a polynomial number of processors. The problems solvable within these limits belong to the class *NC*. *NC* is an acronym for Nick (Pippenger)'s Class.

Many problems that belong to class *P* are also known to belong to *NC*. Multiplication, division, sorting and other very well known problems have all been solved in polylogarithmic $O(\log^{O(1)} n)$ parallel time (see for example [GR88], where various problems are treated).

Unfortunately, and as for the case of problems in *NP* for which no polynomial time sequential algorithms have yet been found, some problems in *P* seem to be very difficult to solve in polylogarithmic parallel time. A problem of this kind is finding the greatest common divisor (*gcd*) of two integers. Although Euclid found a polynomial time 'sequential' algorithm for it 2300 years ago, no one has yet managed to find a way to speed up the computations involved [H87].

This implies the fact that problems in *P* cannot all be claimed to belong to *NC*, whereas the opposite is true, implying (although no concrete proof exists) that the classes *P* and *NC* are distinct.

Chapter 2

Techniques for efficient problem solving on SIMD computers

2.1 Introduction

The advent of parallel computers, whether theoretical or practical has led to the emergence of new techniques and paradigms for solving problems with the goal of achieving optimal and/or efficient bounds defined by many criteria. Efficiency bounds depend on the computational environment. Within the P-RAM model, the class *NC* defines efficiency. For shared memory models, the network diameter provides a natural lower bound on computation time. For instance on a $(\sqrt{n} \times \sqrt{n})$ mesh, a useful definition of an efficient algorithm is an algorithm that runs in $O(\sqrt{n})$ parallel time for a problem of size n. For the hypercube family and networks of diameter $= O(\log n)$, algorithms are efficient if they run in $O(\log n)$

time. For our purposes we will also define an algorithm to be nearly efficient if it runs within a $\log^k n$ (for some integer k) of the diameter of an architecture.

Moreover, a parallel algorithm is called optimal ([GR88], [KR91]) if the product P =(parallel running time $\times$ number of PE's used) is a linear function of the input size or equal to the running time of the best sequential algorithm that solves the same problem.

Our goal in this chapter is to describe a set of techniques that differ in nature and to highlight for some the issues related to their future use in later chapters. At first, emphasis will be put on those which are qualified as algorithmic techniques such as the balanced binary tree, the doubling technique and others. Most of these rely extensively on the tree structure which in parallel computation appears in many facets. Then techniques that help tackle or incorporate solutions to the important issue of communication overheads are presented. These appear under the headings efficient data distribution and non-conventional input schemes. Finally, the technique of embedding structures (such as trees) on interconnection networks and that of hardware enhancing are introduced.

2.2 The balanced binary tree method

Although chapter 5 is almost entirely devoted to the balanced binary tree method on feasible SIMD computers to solve many problems efficiently, the aim of this section is to present it in a P-RAM context. Issues related to the implementation of algorithms where this method is applied on more feasible machines (the same is valid for all the algorithmic methods) are highlighted.

This method makes use of a constructed balanced binary tree. Internal nodes store the result of subproblems with the root corresponding to the global problem. Solutions to problems structured in this way are found in a bottom-up fashion with those at the same level of the tree being computed (combined) in parallel. For instance problems involving the computations of quantities such as $A(0) \oplus A(1) \oplus A(n-2) \oplus \ldots \oplus A(n-1)$ over an array $[A(0), \ldots, (n-1)]$ where $\oplus$ is a binary associative operator and n is a power of two (otherwise a minimum number of neutral dummy elements are added) are best achieved using this method [GR88].

On a P-RAM the above quantity is computed in $O(\log n)$ parallel time with a maximum number of $n/2$ processors (using $A(1)$ to store the final result) by executing the following :

```
$d \leftarrow (n-1)/2$
repeat until $d = 0$
begin
for all $j$, $0 \leq j \leq d$ in parallel do
$A(j) \leftarrow A(2j) \oplus A(2j+1)$
$d \leftarrow (d-1)/2)$
end
```

Some algorithms might use the balanced binary tree technique just to compute some partial results. The implementation or design of such algorithms on feasible SIMD computers will undoubtedly depend in the first place on the nature of the computations to be performed after the construction of the tree. These computations requiring a resident constructed tree in the architecture will have a variety of data exchanges requirements (as will be seen in later chapters). Thus necessitating the application of other techniques to satisfy these requirements in the best possible way.

2.3 The compression technique

The aim of using the compression technique is to recursively reduce a set of entities acted upon by a factor of 2 by performing required computations. In its simplest manifestation this technique is applied to data structures such as, for instance, arrays where two entries can be compressed into a single one. This is in some cases equivalent to the use of the balanced binary tree method. But the non-trivial power of compression is highlighted in many graph algorithms where it is referred to as the vertex collapse technique. Amongst these are the ones that find the connected components of a graph where the strategy is to reduce sets of vertices into supervertices (see [QD84]). On a P-RAM, compression usually leads to $O(\log n)$ solutions because of the absence of communications overheads. But on distributed memory machines such as feasible SIMD computers where data is spread across the network, this is not always implemented in a straightforward fashion. Chapter 4 is partly devoted to the use of the compression technique on feasible SIMD computers where various problems are treated.

2.4 The tree contraction technique

The tree contraction technique was initially designed to evaluate arithmetic expression given in a tree form [MR85], [GR88], but since then it has found a much wider applicability [KR91]. This technique is that of shrinking a binary rooted tree with an irregular height by recursively computing internal nodes. If a tree of size n has height $\log n$, then this method is equivalent to the compression technique. In chapter 4 we show that a P-RAM expression evaluation algorithm (using this method) can be efficiently implemented on some feasible computers.

2.5 The 'divide and conquer' technique

The way of proceeding when using the very well known technique of 'divide and conquer' ([AHU74], [GR88], [K85]) is to divide a given global problem into a number of independent subproblems and then to solve these in a recursive manner. The depth of the recursion is an important factor when adopting this technique as it determines the parallel running time for solving the overall problem.

At any level of the recursion the solution to anyone problem is found independently (from problems on the same level) by combining solutions to its subproblems. On a P-RAM, if subproblems have each a size which is at least a fixed proportion of the problem they compose, then the depth is logarithmic.

Formally, the divide and conquer strategy has the following recursive structure. Given a problem P :

```
If $P$ is decomposable into smaller problems
then
    Divide $P$ into two or more parts ($P_1$, $P_2$, ..., $P_n$)
    In parallel
        Solve $P_1$
        Solve $P_2$
        ...
        Solve $P_n$
        Combine the partial solutions to obtain a solution to $P$
else
    solve $P$ directly.
```

The use of the divide and conquer technique on feasible SIMD machines is also treated in chapter 4 (along with compression and tree contraction) since all the algorithmic techniques presented are not entirely disjoint.

2.6 The doubling technique

The doubling technique is usually applied to data structures such as 1-dimensional arrays and lists. A necessary definition before briefly describing this technique is that of the distance between two elements in these data structures. In a 1 dimensional array this distance may be thought of as the difference between the indices of two elements and in a list of it represents the number of pointer jumps from one element to another.

The doubling technique proceeds by a recursive application of the required calculation to all elements over a certain distance (in the data structure) from each individual element. This distance is doubled at each iteration. For arrays or lists of length $n = 2^k$, a P-RAM computation using the doubling technique will be completed for each element after $k = \log n$ stages. The implementation of the doubling technique for performing some computations such as ranking the elements of a list on computers such as the MCC^2 is also treated in chapter 4.

2.7 Efficient data distribution

Data distribution can have a significant effect on the amount of execution time of parallel algorithms since adopting the right data distribution has proved to improve on communication costs and consequently on execution times. Data distribution is affected by the number of PE's available or the mapping of data items to PE's at the start of a computation. Although unbounded parallelism is allowed, it is often the case that on P-RAM's fewer PE's than data items are used to solve a problem and this within the same time complexity as if it was to be solved with a number of PE's equal to the input size.

A typical example on which this way of proceeding is best illustrated is the computation of quantities such as $Q = A(1) \oplus A(2) \oplus A(3) \oplus \ldots \oplus A(n)$ over an array $A = [A(1), \ldots, A(n)]$ by the use of the balanced binary tree. As seen earlier a P-RAM algorithm can compute this quantity in $O(\log n)$ parallel time with $p \leq n/2$ PE's.

The idea behind reducing the number of processors comes from the fact that at each iteration of our algorithm (equivalent of climbing the balanced binary tree) the number of PE's used is reduced by a factor of 2 making a high proportion of them become idle. One way to reduce such an effect is that $p < n/2$ PE's can be used and still lead to an $O(\log n)$ solution. The strategy is to partition the n elements of the given array into p groups (every PE will be in charge of 1 group) where $p-1$ of them will contain $\lceil n/p \rceil$ elements and the remaining group will only contain $(n - (p-1)\lceil n/p \rceil)$ elements. All the p PE's in parallel then compute a quantity similar to Q within their assigned group in a sequential manner. For any group the maximum number of computations of the type $A(i) \oplus A(j)$ is bounded by $\lceil n/p \rceil - 1$ implying that the problem of computing quantities such as Q (of size n) could be reduced to a problem of size p in $\lceil n/p \rceil - 1$ time units. This newly created problem is then solved using the balanced binary tree method in $O(\log p)$ parallel time.

Thus, the overall computation of Q can be achieved in $\lceil n/p \rceil - 1 + \log p$ parallel time using $p < n/2$ processors. If $p = n/\log n$, then this is done in an optimal fashion. A more general approach to this problem is the application of Brent's theorem which states that if an algorithm A has a parallel running time of t and if A involves a total number of l computations, then A can be implemented using p processors in $O(l/p + t)$ time. [GR88] contains a simple proof of Brent's theorem.

Incidentally this same approach is sometimes forced to be adopted in real parallel environments and has proved to be extremely useful as it drastically reduces inter-processor communication [AL81]. For example the cost of external sorting is very expensive and thus one would have to adapt existing sorting algorithms (based on element per PE) and to assume the availability of enough memory space to store a reasonable amount of data. A typical example is the k-fold bitonic sort algorithm of Hsiao and Shen [HS85] who adapt Batcher's bitonic sort [B75] on the MCC^2 to sort sequences containing more elements than the number of PE's available.

In the case of a one-to-one mapping of input data items to the PE's of a parallel computer, we consider such a mapping to be efficient if it allows the design of efficient algorithms. On a P-RAM we do not need to worry as on this model all mappings are equivalent with regard to communication costs. However, on machines such as the MCC^2, this issue is sometimes fundamentally important in the design of efficient algorithms. The PE's of a $(\sqrt{n} \times \sqrt{n})$ MCC^2 can be indexed according to many indexing schemes which are one-to-one mappings from the coordinate space $\{0, 1, \ldots, \sqrt{n}\} \times \{0, 1, \ldots, \sqrt{n}\}$ onto the index space $\{0, 1, \ldots, n-1\}$ each having properties that makes it suitable for particular applications. Figure 2.1 shows the four most popular indexing schemes of a MCC^2.

The row-major indexing or identity indexing (Figure 2.1.a) which is based on a top to bottom, right to left ordering seems to be the most natural way of indexing the PE's of a mesh but appears only to be suitable for computations with very low inter-processor communication [MS88] such as for instance, the

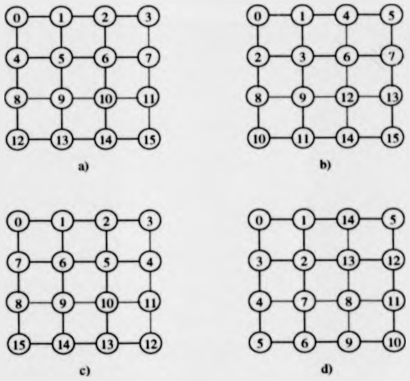


Figure 2.1

very simple problem of adding n numbers. Problems with high inter-processor communication such as sorting do not perform well on the MCC^2 with such an indexing. The algorithm in [O75] requires $O(\sqrt{n}\log n)$ parallel time to sort a sequence of length n due to such an indexing.

Another indexing which has received much attention due to its properties is the shuffled row major indexing (Figure 2.1.b). Such an indexing which is based on the recursive division of the PE's of the mesh into quadrants can be useful in designing algorithms based on the 'divide and conquer' strategy and thus involving a great deal of communication between PE's.

Two other indexing schemes to be used on a mesh are the snake-like order and the proximity order (Figures 2.1.c and 2.1.d). The former which has the property that successively indexed PE's are adjacent has also proved to be very useful. For instance, Schnorr and Shamir [SS86] use such an indexing to design a very simple efficient algorithm for sorting on the (MIMD) MCC^2. The latter indexing combines the advantages of some of the other indexings and is based on space filling curves. Like the shuffled row major order, this indexing recursively divides the mesh into quadrants and like the snake-like order, successively indexed PE's are adjacent. Miller and Stout [MS89] use such an indexing to design efficient algorithms for a wide range of problems in computational geometry.

In later chapters when trying to solve a variety of problems, other advantages and shortcomings of indexing schemes such as the ones described in this section will be highlighted.

2.8 Non-conventional input schemes

Another way for easing difficulties such as communication overheads when solving problems on feasible parallel computers is to use non conventional input schemes. In these schemes no data is stored in the PE's memory at the start of a computation but is rather input gradually maintaining a balance between communication and computation.

A very simple algorithm to use such a scheme is the algorithm of Guibas et al. [GKT79] to compute the transitive closure of a directed graph. In this algorithm (Boolean) matrix multiplication is achieved in $O(\sqrt{n})$ parallel time

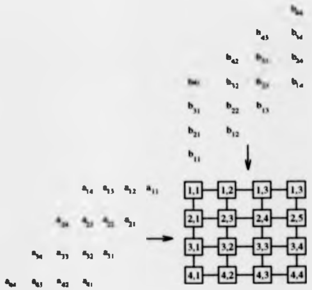


Figure 2.2

(for two $(\sqrt{n} \times \sqrt{n})$ matrices on a MCC^2 with $(\sqrt{n} \times \sqrt{n})$ PE's) by the use of a skewed input scheme as shown in Figure 2.2 (for two matrices denoted A and B). At each step of the computation, matrix A is pushed one step to the right and matrix B is pushed one step down, and each PE (identified in this case by its geometric coordinates (i, j)) multiplies the values it receives (a_{ij} and b_{ij}) and adds the result to an accumulator. After precisely $2\sqrt{n} - 1$ steps every PE(i, j) will contain the required value $(\sum_{k=1}^{\sqrt{n}} a_{ik}b_{kj})$.

Amongst other algorithms that use a similar kind of input scheme and which can be regarded as a sort of data pipelining, is the $O(\sqrt{n})$ algorithm of Maggs and Plotkin [MP88] for finding the minimum-cost spanning tree of an n-vertex undirected graph on a $(\sqrt{n} \times \sqrt{n})$ MCC^2.

2.9 Graph embeddings

Another general approach for efficient problem solving is the embedding of structures such as graphs (in particular trees) in interconnection networks. Such embeddings are of a great interest in simulation studies. For example, embedding trees in some interconnection networks may efficiently simulate a P-RAM algorithm based on such structures. Moreover, inter-mapping of topologies on which interconnection networks are based would provide a view on how efficiently a particular network might simulate another [U84].

In graph theoretical terms, an embedding of a graph G (called the *guest* graph) into another graph H (called the *host* graph) is a mapping of the edges of G into paths of H such that each vertex of G maps to a single vertex of H. The quality of an embedding is usually measured by three parameters:

a) dilation which is equal to the maximum length of any path in the host graph to which an edge of the guest graph is mapped.

b) expansion which is the ratio of the number of nodes of the host to the number of nodes in the guest.

c) congestion which is the maximum number of paths (mappings of the guest graph) using any edge of the host graph.

Considering only embeddings where at most one node of the *guest* is associated

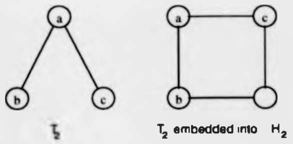


Figure 2.3

with any single node of the *host* and labeling the *dilation*, the *expansion* and *congestion* respectively as d, e, and c, we note for instance from [Gi91] that the double rooted binary tree embeds in the hypercube (with the same size n) with $d = 1$, $e = 1$ and $c = 1$, and that any mesh with n nodes whose dimensions are each a power of two is a subgraph of its optimum hypercube. By optimum hypercube, we mean the smallest possible hypercube with n' nodes such $n \leq n'$ Ullman [U84] describes embeddings of complete binary trees and other graphs in a VLSI context.

In what follows we describe how the complete binary tree T_n (of height n) with $2^n - 1$ nodes can be optimally embedded in the hypercube of dimension n with 2^n nodes (denoted H_n) following Wu's method [W85]. For $n = 1$, it is trivial to see that T_1 can be embedded in the hypercube of dimension 0 (H_0). For $n > 1$ there is no embedding of T_n into the hypercube H_n with d (*dilation*) = 1 unless $n = 2$. Thus, T_2 can be embedded in the 2-dimensional hypercube with $d = 1$ and $c = 1$ as shown in figure 2.3.

From [W85], a complete binary tree of height $n > 2$ can be embedded into a hypercube of dimension $n + 1$ (H_{n+1}) with $d = 1$ and into a hypercube H_n

with $d = 2$. The second claim is proved by showing first that there exists no embedding with dilation = 1 and by stating two properties that the embedding with dilation = 2 must satisfy to finally prove by induction that these properties hold for all values of n. These properties are called the *cost2 Property* and the *Free Neighbor Property*.

1) Cost2 Property : If A is the root of T_n and L and R are respectively the roots of its left and right subtrees, then the distance between the vertices that A and L are mapped to in the n-dimensional hypercube H_d is 2 while that between the vertices that A and R are mapped to is 1.

2) Free Neighbor Property : The only free (no vertex of the binary tree is mapped onto it) node in the hypercube is a neighbour to the node to which the root of T_n is mapped.

From an embedding with $d = 2$ of a binary tree T_{n-1} into a hypercube of $n-1$ dimensions (H_{n-1}) that verifies the two above stated properties, Wu [W85] obtains an embedding of T_n with $d = 2$ into the n-dimensional hypercube H_n that will also verify these properties by the following construction.

i) Embed the left subtree of T_n into $0H_{n-1}$ ($0H_{n-1}$ denotes the dimension $n-1$ of H_n comprised by the vertices (PE's) whose most significant bit is 0). Let $0A$ be the vertex in H_n to which the root of the left subtree is mapped to and let $0B$ be its free neighbour.

ii) Embed the right subtree of T_n into $1H_{n-1}$ ($1H_{n-1}$ denotes the dimension $n-1$ of H_n formed by the vertices (PE's) whose most significant bit is 1). Let $1A$ be the vertex in H_n to which the root of the right subtree is mapped to and let $1B$ be its free neighbour.

iii) Map the root of H_n to the vertex (PE) $1B$.

Again we have given a flavor of the technique of embedding graphs because we rely on such a technique to show (in chapter 5) that it is important in the efficient implementation of other techniques (such as the balanced binary tree) on feasible SIMD computers.

2.10 Augmenting architectures

Another technique for speeding up applications on parallel computers is the addition of extra hardware or features to a particular architecture. This consists of adding extra simple connections between PE's, extra PE's and connections or buses for conveying data in a faster fashion. The issue of adding extra connection to an architecture is best illustrated on the 2-dimensional mesh connected computer which can be enhanced by the so-called 'wrap-around' (figure 2.4(a)) or 'toroidal' (figure 2.4(b)) connections [HB85]. Unfortunately this type of enhancement will only help reduce time complexity terms (for certain problems) by constant factors due to the fact that it only speeds up communications by offering shortcuts within the architecture. In the case of the MCC^2 with 'wrap-around'

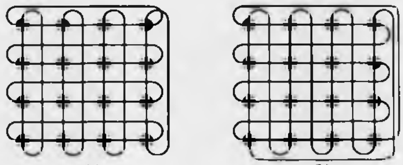


Figure 2.4

connections the diameter is halved. The addition of extra connections might add to the properties of an architecture. In chapter 4, we show that an augmented perfect-shuffle computer can support some P-RAM algorithms in a better way than its non-augmented counterpart.

Augmenting architectures by means of extra processing elements can also be illustrated on previously mentioned architectures. The 'Cube-Connected Cycles' computer proposed by Preparata and Villeumin [PV81] can in a way be regarded as opting for augmenting the cube-connected or hypercube computer by means of extra processing elements. They showed that if every PE in a CCC is replaced by a cycle consisting of a fixed number of PE's, then the result is a very powerful interconnection network. The same thing could be said about the q-dimensional Mesh Connected Computer which can be considered as the superimposition of q 2-dimensional MCC's, a structure that has attracted many by its topological simplicity.

Chapter 3

Tools for efficient problem solving on SIMD computers

3.1 Introduction

The continually growing body of parallel algorithms has undoubtedly highlighted the importance of many paradigms. In the previous chapter we described a variety of techniques ranging from algorithmic to hardware enhancing. Our purpose here is to review some primary algorithms (or library candidate routines) and other utilities that have proved very useful in the design of parallel algorithms.

A logical consequence of identifying such primary algorithms and utilities is the establishment of a structural consistency amongst parallel algorithms, at least for those that relate to a common domain. Vishkin [Vi91] illustrates such a concept very elegantly by compiling different structures for many types of problems and by stressing the usefulness of many simple algorithms and utilities.

For instance, for some list, tree and graph problems, the structure compiled (of which an extract is shown in figure 3.1) is for problems whose solutions on the P-RAM model of computation are known to be in *NC*. This structure shows that solutions to problems or utilities higher in the diagram incorporate solutions to problems or utilities lower in the diagram. As an example the solution to the prefix sums problem (bottom of figure 3.1) is a key subroutine in the solution to the list ranking problem which is itself a key subroutine used the utility called the ***Euler tour*** technique. Further up, this structure shows that the technique of tree contraction or the problems of finding the lowest common ancestors (lca's) and graph connectivity make use of the Euler technique.

On feasible interconnection networks, Vishkin's structure [Vi91] holds from a relational point of view. That is, one can always solve any problem as on the P-RAM model of computation using the same primary algorithms and utilities but with modified time complexities. For instance on the MCC^2 (with a diameter of $2\sqrt{n}$) it will take at least $O(\sqrt{n})$ time to solve each problem in figure 3.1. Communication overheads are closely responsible for this lower bound on this computation time. Solutions to the communication or routing problem must be therefore included in such a structure because assumptions about the execution of algorithms on interconnection networks becomes worthless without their inclusion. Another important problem related to the communication problem and which should also be included in such a structure is sorting.

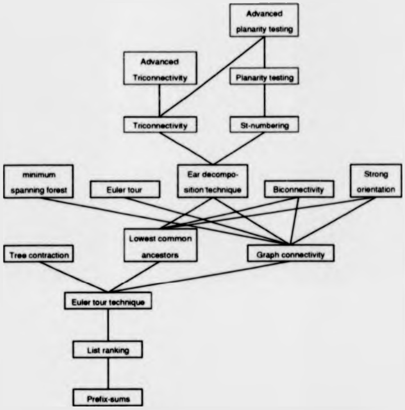


Figure 3.1: List, tree and graph problems linkage

In what follows, along with sorting and communication strategies on feasible machines we look at a set of primary algorithms and utilities that have greatly facilitated the design of algorithms on the P-RAM model. These are the prefix computation problem, the Euler tour technique and the ear decomposition technique.

3.2 Sorting on SIMD computers

Sorting on a parallel computer with n PE's, is the problem of reordering a set of n keys so that at the end of the computation the i^{th} smallest key is stored at the i^{th} PE. Understandably it is one of the problems that has attracted a lot of interest within the community of parallel algorithms researchers. This has led to numerous solutions on various models of parallel computation which range from the implementation of known sequential sorting algorithms to a variety of new concepts (see for instance [A85]).

For the P-RAM model of computation, many sorting algorithms have been proposed (see for example [C86], [BH82], [SV81] and [P78]). The latest result is the $O(\log n)$ (optimal) algorithm of Cole [C86] to sort a sequence of n items using n processors. Implementing P-RAM sorting algorithms or designing new ones on many realistic machines within optimal bounds has often been prohibited by the high inter-processor communication requirements of the sorting problem.

Instead solutions based on circuit comparators such that of Batcher [B68] have been proposed for feasible SIMD machines. For instance Nassimi and Sahni

[NS78] and Thompson and Kung [TK77] mapped Batcher's bitonic sort in $O(\sqrt{n})$ on the q-dimensional mesh-connected computer. Similarly it was implemented on n PE's architectures such as the hypercube [RS90], the perfect-shuffle [S71] and the cube-connected cycles architectures [PV81] in $O(\log^2 n)$ parallel time.

Unlike the case for the MCC, the question of sorting on these architectures in times proportional to their diameters (i.e $O(\log n)$) is still open. More recently Cypher and Plaxton [CP90] obtained an $O(\log n(\log\log n)^2)$ for sorting a sequence of n elements on an n-PE's hypercube.

3.3 Routing on SIMD computers

A major problem in parallel computing on distributed memory machines is how to organise communication through the interconnection network for data exchanges between the processors. Frequently, the cost of routing is the dominating term in the time required to solve a problem on such machines. Amongst others, the algorithm of Thompson and Kung [TK77] for sorting a sequence of n numbers on a $(\sqrt{n} \times \sqrt{n})$ mesh connected computer, where the PE's are indexed according to the shuffled row major order, uses only $O(\log n)$ comparison steps but has $O(\sqrt{n})$ routing steps. Solving the routing problem also arises when trying to simulate particular operations of a machine such as the P-RAM on more realistic machines.

3.3.1 The routing problem

Depending on the application, routing on a parallel computer can occur in different forms as defined by Ullman [U84] :

Permutation routing occurs when each PE requests access to the memory of another distinct PE. *Partial routing* occurs when each PE of a proper subset uniquely accesses a memory location. In *many-one routing* each PE requests an access to some memory and many PE's may request access to the same memory. As some requests could be nullified, this form of routing is seen as a generalisation of *Partial routing.*

Formally the routing problem on an n PE's parallel (distributed memory) model of computation could be stated as follows : Each PE with index $i \in I = [0..n-1]$, initially contains an address $a(i) \in \{0, 1, \ldots, n-1\} \cup \emptyset$ of another destination PE. The communication requirement to satisfy depends on the specification of the elements of the set A:

It is a permutation routing iff : $\forall i, j(i \neq j) \in I$ we have $a(i) \neq a(j) \neq \emptyset$. It is a partial routing iff : $\forall i, j(i \neq j) \in I$ we have $a(i) \neq a(j)$ provided $a(i) \neq \emptyset$, $a(j) \neq \emptyset$ and it is a many-one routing iff : $\forall i \in I$ we have $a(i) \neq \emptyset$.

3.3.2 Simulation of P-RAM's

Finding solutions to the routing problem in its various forms on distributed memory machines allows the simulation of read and write operations of ideal parallel computers such as the P-RAM. For instance, solutions to permutation routing can simulate the $EREW$ P-RAM since it does not allow either concurrent reads nor concurrent writes. The more powerful $CRCW$ can be simulated by a solution to the *many-one* form of routing on condition that arbitration rules are set to resolve writing conflicts [U84].

Moreover, there are two ways for solving all forms of the routing problem. The first is to proceed deterministically and the second is to use randomness. The difference between the two methods lies in the way of choosing intermediate PE's when messages are routed. The next two sections outline solutions from both methods. Having to deal with the SIMD class of computers, we are therefore restricted to the instance of the problem where all the requests made by the processors are done in a synchronous fashion.

3.3.3 Deterministic routing on feasible architectures

In deterministic routing, a message is wholly directed from a source PE to a target PE via other PE's chosen deterministically.

Permutation routing can be reduced to sorting and therefore messages can be routed using *compare-exchange* and *near-neighbour* routing operations. It has efficient deterministic solutions on many architectures. On the 2-dimensional MCC, the algorithms of Nassimi and Sahni [NS81] and Thompson and Kung [TK77] that run in $O(\sqrt{n})$ parallel time for n-items permutations serve our purpose. Further algorithms that also perform the same task in the same time order have been designed to improve on the constant factor of the leading complexity term [SS86]. On architectures such as the (n PE's) CCC or PSC, the results of [RS90], [S71] reported in the previous section for solving the n-items permutations achieve our goal on these machines but only in $O(\log^2 n)$ parallel time. The desired complexity is of course $O(\log n)$ which is proportional to the diameters of such architectures, but the only known method is to use randomness.

In the following section, an outline of a deterministic solution to the many-one form of the routing problem is presented. Moreover, such a solution encompasses solutions to all the other forms. The algorithm which is extensively used in this thesis as a library routing procedure on the MCC^2 is due to Nassimi and Sahni [NS81]. It achieves the goal of running, within a constant factor of the optimum time of $2\sqrt{n}$. This algorithm was also designed for machines such as cube connected (CCC) and perfect shuffle (PSC) computers and runs in $O(\log^2 n)$ on such networks of size n. The $O(\log n)$ (probabilistic) algorithms of Valiant and Brebner [VB81] (section 3.3.4) and Aleliunas [Al82] are rather preferred for these architecures to execute partial or permutation routing.

Nassimi and Sahni's routing algorithm

The algorithm of Nassimi and Sahni [NS81] allows the simulation of concurrent read and write operations of the P-RAM model of computation on more realistic machines by the use of the techniques of compacting and replicating data.

Nassimi and Sahni [NS81] identify the routing problem in two forms. The first is called the *Random Access Read* (RAR) and occurs when a PE wishes to acquire a data item from another PE, not necessarily a direct neighbour. The second is called the *Random Access Write* (RAW), and occurs when a PE wants to send (transmit) a data item to another processor.

RAR and RAW require some well defined subalgorithms (procedures) called *sort*, *rank*, *concentrate*, *distribute* and *generalise*. These, manipulate records containing data to be routed as well as other routing information. They can be briefly described as follows :

Sort This procedure simply sorts a sequence of records ($G(i)'s$) held by the PE's of the MCC in non-decreasing order on the key target which is the address to which data is to be sent to, or read from. If $H(i)$ is the key target then after an application of **sort**, records will be rearranged so that :

$$H(i) \leq H(i+1),\ 0 \leq i \leq N-1\ \ (N\ :\ total\,number\,of\,PE's)$$

Again, Nassimi and Sahni's [NS79] and Thompson and Kung's [TK77] sorting algorithms are amongst the known algorithms to achieve this task in a strict SIMD context.

Rank The objective of rank is to assign to each selected record held by a PE a rank which is the number of selected records held by other PE's having a smaller index. A record is selected if it is held by the PE with the highest index amongst the (sorted) set of PE's requesting the same address. Suppose we have the following set of records:$(a, b, c, c^*, a, a^*, e^*, f^*)$ (a starred value denotes a selected record), then the output of **rank** is $(-, -, -, 0, -, 1, -, 2, 3)$

Concentrate The main goal of procedure concentrate is to displace the ranked records to the PE's whose indices equal the ranks computed in the previous step. Let $G(i_r)$ $(0 \leq r \leq j < N)$ be a set of records initially stored in PE(i_r) and assume that these records have been ranked so that $H(i_r) = r$. A **concentrate** results in record $G(i_r)$ being moved to PE(r).

Distribute Distribute is the inverse of procedure concentrate. Its purpose is to move records to the PE's whose indices equal the addresses carried by these records. Let $G(i)$ $(0 \leq i \leq i < N)$ be a set of records with $G(i)$ in $PE(i)$. Let $H(i)$ $(0 \leq i \leq j)$ be a set of destinations such that $H(i) < H(i+1)$ $(0 \leq i \leq j)$. Distribute routes $G(i)$ to $PE(H(i))$ $(0 \leq i \leq j)$.

Generalise The purpose of generalise is to copy (replicate) a record held by a PE with an index equal to the rank of this record into all the PE's whose indexes are less or equal to the address carried by this record. let $G(i)$ $(0 \leq i \leq j < N)$ be stored in PE(i). Each record has a field H such that $0 \leq H(0) \leq H(1) \leq \ldots \leq H(j) \leq N - 1$ and $H(i) = \infty$ $j < i < N$. **Generalise** makes copies of record $G(i)$ in PE$(H(i-1)+1)$ through PE$(H(i))$ $0 \leq i \leq j$. $H(-1) = 0$.

An *RAR* (simulating concurrent reads) is performed using (in order) **sort, rank, concentrate, distribute, concentrate, generalise** and finally **sort**.

The *RAW* problem (simulating concurrent writes) is simpler to deal with than the RAR. When no two PE's are sending data to the same PE then **sort** followed by **distribute** will achieve our purpose. In the event where many PE's have the same target PE, two cases are distinguished : Either only one PE is made to succeed and thus, an arbitration rule among contending PE's has to be set, or all requests to write are to be honored which results in compacting data from all PE's. In both cases, the ordered sequence of subalgorithms to perform is **sort, rank, concentrate** and **distribute** with the difference that dissimilar records are manipulated for each case.

With all factors considered (including d which is the maximum number of data items to be written into any one PE), the overall time complexity of executing RAR's and RAW's on a q-dimensional MCC is respectively $O(q^2n^{1/q})$ and $O(q^2n^{1/q} + dqn^{1/q})$. For an n PE's PSC or CCC, the time complexity of performing a RAR is $O(\log^2 n)$ and it mounts up to $O(\log^2 n + d\log n)$ for executing a RAW.

3.3.4 Randomised routing

In randomised routing data is forwarded between a source PE and a target PE via intermediate PE's chosen at random. The algorithm of Valiant [V80] was the first algorithm to realise partial and permutation routing on a cube connected computer (CCC) with n processors in only $O(\log n)$ parallel time. It performs well on other interconnections also [VB81]. The strategy employed is a two-phase strategy and consists of first sending data from each PE (involved in the communication requirement) to another PE chosen randomly and then to send the data to their true destinations.

Valiant and Brebner's routing algorithm

A high level description of this algorithm can be stated in two steps as follows:

1. For each PE(i) that wishes to send a data item (packet) to another PE(j), select randomly another index k by picking each of the n bits in the binary representation of k to be 0 or 1, independently, each with probability 1/2 and following a *left-to-right* routing strategy send (transmit) the data item to PE(k). In the i^{th} step of a left to right strategy the data is routed so as to to correct the i^{th} bit (from the left) of the current address of each datum compared with its destination. In case of competition for a wire (connection) to leave a PE, packets are queued and transmitted one at each step. The priority is given to the packet with the farthest destination.

2. The packet, say from PE(i) having reached PE(k), is given again a left-to-right route to its true destination PE(j) in a similar manner.

The time complexity of Valiant and Brebner's algorithm is $O(\log n)$ with overwhelming probability for both the *PSC* and *CCC* architectures with n PE's. More precisely Valiant [V80] has shown that for the hypercube with n PE's the probability that messages will take more than $8 \log n$ time to be routed is less than $(0.74^{\log n})$. This probability converges towards zero exponentially with the dimension of the architecture.

3.4 Prefix sums

The prefix sums or prefix computation problem is an important problem in various fields (see e.g [Ki90]. For ease of description the problem is described as follows [GR88]: Given an associative binary operator $\odot$ (e.g. min, max, $+$, $\times$) and an array $[A(1), A(2), \ldots, A(2n-1)]$, compute $A(n)$, $A(n) \odot A(n+1)$, $A(n) \odot A(n+1) \odot A(n+2) \ldots$, using the locations $[A(1), A(2), \ldots, A(n-1)]$ to store intermediate results .

We describe below a P-RAM algorithm which uses the balanced binary tree method and runs in $O(\log n)$ time with n PE's. The leaves of the tree initially contain the values $(A(n), A(n+1), \ldots, A(2n-1))$. An auxiliary vector B is also needed to store intermediate and final results.

There are two phases in the computation. The first consists of constructing the balanced binary tree. This takes $\log n$ steps after which, every non-leaf node contains $(value(ls) \odot value(rs))$ (ls: left son, rs: right son). Results are stored in locations $A(1)$ through $A(n-1)$. This phase is described as follows:

```
**for** $k = (\log n) - 1$ **step** $-1$ **to** $0$
**for all** $j$, $2^k \le j \le 2^{k+1} - 1$ **in parallel do**
                    $A(j) \leftarrow A(2j) \odot A(2j+1)$
```

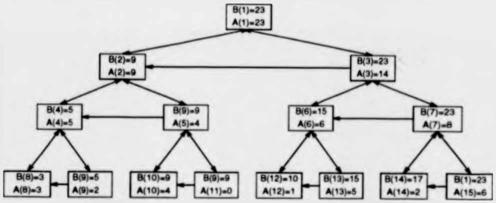


Figure 3.2: Prefix computation on a P-RAM

The second phase is a top-down phase also taking $O(\log n)$ time. In the case where a node is a right son it will store the A value of its father, and ($value\ of\ its\ father \ominus value\ of\ its\ brother$) otherwise, where $\ominus$ is another binary operator adequately chosen. Usually, this binary operator denotes the reverse operation. This second phase is described as follows:

```
$B(1) \leftarrow A(1)$
for $k = 1$ to $\log n$ do
for all $j$, $2^k \leq j \leq 2^{k+1} - 1$, in parallel do
                    if $j$ is odd then $B(j) \leftarrow B((j-1)/2)$
                                   else $B(j/2) \ominus A(j+1)$
```

Figure 3.2 shows an example of the P-RAM prefix computation algorithm (using the balanced binary tree method) run on a sequence of eight elements and with the operator $\odot = +$. Arrows represent the data exchanges between the processors.

3.4.1 Implementation of the P-RAM prefix computation algorithm on the 2-dimensional mesh-connected computer.

A straightforward implementation of the P-RAM prefix sums algorithm just presented would also consist of two phases. For an array $[A(0), \ldots, A(n-1)]$, the computation (with a binary operator $\odot$) would require the use of a constant number of additional registers per PE (registers B, C, D). The syntax of the algorithm performing this step is different from that on the P-RAM because the same formulation of the problem would have required the use of an $(\sqrt{2n} \times \sqrt{2n})$ MCC^2 to store the array $[A(1), A(2), \ldots, A(2n-1)]$. Using the convention that $A(i)$ is stored at $PE(i)$ at the start of the computation, then the following completes the first phase.

Phase 1

```
for all $i$, $0 \le i \le n/2-1$ in parallel do
        begin
        $B(n/2+i) \leftarrow A(2i) \odot A(2i+1)$;
        $C(n/2+i) \leftarrow A(2i+1)$
        end
for $k = m-2$ step $-1$ to 0 do ($m = \log n$)
for all $i$, $2^k \le i \le 2^{k+1}-1$ in parallel do
        begin
        $B(i) \leftarrow B(2i) \odot B(2i+1)$;
        $C(i) \leftarrow B(2i+1)$
        end
```

After this phase the balanced binary tree is constructed with its leaves in $A[0..n-1]$ and its internal nodes in $B[1..n-1]$. We then proceed to the second phase by executing:

Phase 2

```
$D(1) := B(1)$
**for** $k = 0$ **to** $m - 2$
**for all** $i$, $2^k \leq i < 2^{k+1} - 1$ **in parallel do**
                    **begin**
                    $D(2i+1) \leftarrow B(i)$;
                    $D(2i) \leftarrow B(i) \ominus C(i)$
                    **end**

**for all** $i$, $0 \leq i \leq n/2 - 1$ **in parallel do**
                    **begin**
                    $D(2i) \leftarrow D(n/2+i) \ominus C(n/2+i)$;
                    $D(2i+1) \leftarrow D(n/2+i)$
                    **end**
```

At the end of phase 2 the results are stored in the D registers. The syntax of this step is also altered due to the same arguments put in the case of step 1. The complexity of the whole implementation depends of course on the times taken to perform data routings. In chapter 5 we look at the adaptation of the balanced binary tree on feasible machines and show that the P-RAM prefix algorithm can be implemented in a simple way.

3.5 The Euler tour technique

Applied to trees, this new technique can lead to many useful computations as Tarjan and Vishkin [TV85] have showed (see also KR91). Their motivation was the lack of efficient methods to perform some simple computations. For instance using this technique, the problem of finding the number of descendants of each vertex in a tree is reduced to a list ranking problem.

Given an unrooted tree, the Euler tour technique consists of applying two steps which are the replacement of every edge of the tree by two anti-parallel edges (the result of which is an Eulerian digraph) and the computation of an Euler circuit of the newly obtained graph.

Assuming that the tree is given by its adjacency list, the first step is achieved by interpreting the adjacency list as a list of outgoing edges from each vertex. That is, an edge (u, v) will appear in u's list and (v, u) will appear in v's list. The construction of the Eulerian circuit needs at first the preprocessing step of making the adjacency list for each vertex circular i.e. causing the last element to point back to the first. The last element of every list is found by using the doubling technique. The Euler circuit is then found by defining for each edge (u, v) the edge $\mathit{Eulernext}(u, v)$ adjacent to it in the Euler circuit. If $\mathit{next}(u, v)$ is the edge next to (u, v) on the circular list for u, then the following completes the task :

for all (edges) **in parallel do** $Eulernext(u,v) \leftarrow next(v,u)$

3.5.1 Implementation of the Euler tour technique

Implementing the Euler tour technique may be done in the following two ways. For instance, on a $(\sqrt{n} \times \sqrt{n})$ MCC^2, an initial configuration would be to store the adjacency list of every vertex v_i $(i = 0 \ldots n-1)$ in the memory of one PE of the mesh. With this configuration, finding the last element of every list can be done by making every PE search for the last element of the list it holds in a sequential fashion. If d is the maximum degree of our tree then this step is clearly achievable in $O(d)$ sequential time. Finding $Eulernext(v_i, v_j)$ is done by making the PE holding (v_i, v_j) to read (using the RAR procedure of [NS81]) $next(v_j, v_i)$ held by PE(j) (PE(j) also finds $next(v_i, v_j)$ sequentially). Every PE will make at most d such requests which brings the overall time complexity to $O(d^2\sqrt{n})$ parallel time.

The second way of proceeding is to convert the adjacency list of each vertex to a list of edges and to store each edge in the memory of one PE. Atallah and Hambrusch [AH85] showed that with such a configuration the steps of the Euler tour technique can be implemented in $(O(\sqrt{n})$ for a tree with n edges.

3.6 The ear decomposition technique

The *ear decomposition* technique was proposed in parallel environments as a replacement for the *depth-first search* technique (see [KR91], [Vi91]). Since then it has proved extremely useful in designing many efficient graph algorithms.

An ear decomposition $D = [P_0, \ldots, P_{r-1}]$ of an undirected graph $G = (V, E)$ is the partition of the set of edges E into an ordered collection of edge-disjoint simple paths $P_0, \ldots, P_{r-1}$ called *ears*, such that P_0 is a simple cycle, and for $i > 0$, P_i is a simple path (cycle) with each endpoint belonging to a lower-numbered ear, and with no internal vertices belonging to lower-numbered ears [KR91]. An *open ear decomposition* is an ear decomposition in which none of P_i, $i > 0$, is a simple cycle. A graph has an ear decomposition if and only if it is 2-edge connected and a graph has an open ear decomposition if and only if it is 2-vertex connected (biconnected) [KR91]. Briefly the ear decomposition algorithm for an undirected 2-edge connected input graph $G = (V, E)$ can be described by the following :

```
1. Preprocess G:
   1.1 Find a spanning tree T of G;
   1.2 Root T and number the vertices in preorder;
   1.3 Label each non-tree edge by the least common
   ancestor (lca) of its endpoints in T.
2. Assign ear numbers to non-tree edges:
   number non-tree edges from 0 to $r - 1$
```

in non-decreasing order of their lca numbers.

3. **Assign ear numbers to tree edges**:

   number each tree edge with the number of the minimum-numbered non-tree edge whose fundamental cycle it belongs to.

The steps of the ear decomposition algorithm can be implemented in $O(\log n)$ time on the P-RAM model using the Euler tour algorithm together with efficient algorithms for finding a spanning tree, sorting, prefix sums and finding the lowest common ancestors (lca's) for nodes in a graph [KR91].

In the next chapters it is shown that in the case of the MCC^2, efficient algorithms for prefix sums (chapter 5), an Euler tour of a graph and a spanning tree (chapter 6) exist.

Chapter 4

Compression, tree contraction and 'divide and conquer' on feasible SIMD computers

4.1 Introduction

In this chapter we consider a very large class of interesting problems which can be solved on the P-RAM model of computation in $O(\log n)$ time using the techniques of compression, tree contraction and 'divide and conquer'. These problems have the characteristic that they can be solved recursively such that at each recursive step, a problem of size n is reduced to m similar problems each of size $\lceil n/b \rceil + c$ ($b \geq 2$, $c \geq 0$). We shall say that problems that can be so expressed belong to the class R.

Our aim is to show what subclasses of R are efficiently or nearly efficiently solvable on architectures such as the MCC^2, CCC or PSC by implementing their P-RAM solutions. This is equivalent to showing how the techniques of compression, tree contraction and 'divide and conquer' can be used on these architectures. Recall that we defined a solution (algorithm) to be nearly efficient on a feasible machine if it runs within a $\log n$ factor of its diameter. This definition can be relaxed for the CCC and PSC by considering that a solution is nearly efficient on these architectures if it runs within a $\log n$ factor of the time taken to execute some forms of routing.

The specific techniques employed in implementing solutions to the problems in the subclasses of R depend on the parameters m, b and c and particular characteristics of these problems. The approach is to illustrate the methods employed by taking archetypal but simple examples of problems of R for different values of m, b and c. These problems include polynomial evaluation, list ranking and expression evaluation. Also in this chapter, a way for improving the processor utilisation for some problems in R is suggested.

4.2 A simple case

Those problems with parameters $m = 1$, $b = 2$ and $c = 0$ are recursively reducible to a similar problem of half the original size. They can often be solved using a *repeat* statement with a set of instructions embodied in it. These solutions may have the following structure where the block $\{$*instructions*$\}$ contains the instructions leading to a reduction (compression) in size for our problems and the blocks {*initialisations*} and {*reinitialisations*} contain variable initialisation of no great importance to the analysis of solutions. The block {*instructions*} might involve some concurrent reads or concurrent writes, but we do not care since we can simulate these operations using a library routing procedure such as that of Nassimi and Sahni [NS81].

```
{initialisations}
repeat (condition)
        {instructions}
{reinitialisations}
```

A typical example of this class of problems is polynomial evaluation. This problem is that of evaluating at $x = h$, the general polynomial $p(x)$ of degree N where : $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N$ with (for ease of presentation) $N = 2^k - 1$. The polynomial $p(x)$ can be written $p(x) = p'(x) + x^{(N+1)/2} p''(x)$, where $p'(x)$ and $p''(x)$ are similar polynomials of degree $2^{k-1} - 1$. Thus, the following (Algorithm 4.1) provides an iterative evaluation of $p(x)$ at $x = h$ based on the compression technique and which runs in $O(\log N)$ parallel time on the P-RAM [GR88].

```
1. $x \leftarrow h$
2. $d \leftarrow (n-1)/2$
```

```
3. repeat until $d = 0$
4.         if $0 \leq i \leq d$ then begin
5.         $a_i \leftarrow a_{2i} + x a_{2i+1}$
6.         $x \leftarrow x^2$
7. $d \leftarrow (d-1)/2$
8. end
```

Algorithm 4.1

4.2.1 Naive implementation

Implementing polynomial evaluation on a distributed memory machine requires first the mapping of the coefficients a_i's onto the PE's of our architectures. For instance, on a $(\sqrt{n} \times \sqrt{n})$ MCC^2 $(n = N + 1)$ each PE(i) can be provided with one of the constants a_i along with the values of N and h. Within the computation each enabled PE(i) (for $0 \leq i \leq d$ in parallel do ...) recomputes its associated a_i and the values of d and x. At the end of the computation (when $d = 0$) the final result is stored as a_0. Each recomputation of x and d takes constant time and each recomputation of a_i requires the values of coefficients a_{2i} and a_{2i+1}. These can be acquired from their associated PE's by the use of a library routing procedure such as the Random Access Read (RAR) procedure of Nassimi and Sahni [NS81], where each application of RAR takes $O(\sqrt{n})$ parallel time. Since the number of repetitions involved in the repeat statement is $O(\log n)$, an $O(\sqrt{n} \times \log n)$ algorithm has thus been described.

Similar computations on a CCC or a PSC with n PE's run in $O(log^2 n \times \log n)$ time if the routing algorithm of Nassimi and Sahni [NS80] is used or in $O(\log n \times \log n)$ time probabilistically if the routing algorithms of Valiant and Brebner [VB81] and Aleliunas [Al82] areused. Clearly, it can be stated that : *Any $O(\log n)$ P-RAM based the compression technique is naively implemented on more realistic machines in $O(T_r \log n)$, where T_r is the cost of invoking a library routing procedure.*

4.2.2 Some improved results

The above results can be improved on some architectures if the instructions like $a_i \leftarrow a_{2i} + a_{2i+1}$ (line 5 in algorithm 4.1) are carefully looked at. In this respect, the time complexity for polynomial evaluation can be improved on the MCC^2 by a factor of $\log n$. Each iteration of the repeat statement reduces the number of active PE's by a factor of 1/2. If the PE's are taken to be indexed according to shuffled row major order on a mesh, then after each second iteration and due to the instruction $a_i \leftarrow a_{2i} + a_{2i+1}$ the active PE's are made to occupy a square mesh which is 1/4 the original area. Figure 4.1 shows the successive areas occupied by the problem.

The result of this is that during the i^{th}iteration, the cost $O(\sqrt{n})$ of applying the RAR procedure is replaced by $O(\sqrt{n}/2^{(i-1)/2})$ if i is odd and it is replaced by $O(\sqrt{n}/2^{(i/2)-1})$ if i is even. The overall time complexity of the algorithm becomes $\sum_{i=1}^{\log n/2} O(\sqrt{n}/2^{(i-1)} + \sum_{i=1}^{\log n/2} \sqrt{n}/2^{(i-1)} = O(\sqrt{n})$ which within a constant factor

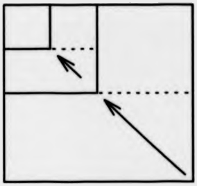


Figure 4.1: compression on a MCC^{*}

is time-efficient and an improvement by a $\log n$ factor.

On a CCC, each iteration of algorithm 4.1 reduces the computation from within one dimension (d) to the next lower dimension ($d-1$) where the number of active PE's is halved. Because the dimension $d-1$ also induces a CCC, then a library routing procedure can be invoked at each i^{th} iteration with a time complexity of $O(\log^k(n/2^{i-1}))$ ($k=1$ *or* $k=2$). This makes the expression that gives the overall time complexity for the polynomial evaluation example equal to:

$$\sum_{i=1}^{\log n/2} \log^k n/2^{(i-1)} = O(\log^{k+1} n) \; (k = 1 \text{ or } k = 2).$$

The situation on the PSC differs from the two previous architectures since its interconnection network is not recursively constructed. A consequence of this is that at each iteration of the repeat statement (in algorithm 4.1) communication between the set of PE's where the reduced problem lies cannot be considered to use just these PE's but must be regarded as using the whole network. That is, it is not possible to invoke a library routing procedure at a reduced time cost. Thus, the solution advocated earlier runs in $O(log^2 n)$ time using probabilistic

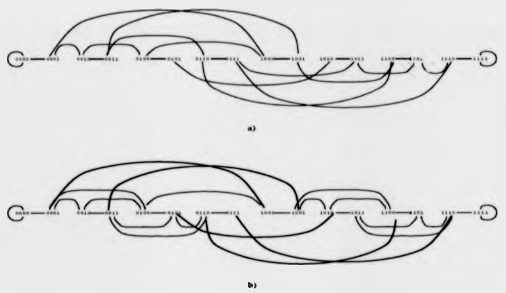


Figure 4.2: (a) A perfect shuffle computer and (b) its modified counterpart

routing or $O(log^3 n)$ time using deterministic routing. These results follow from a summation of the type $\sum_{i=1}^{\log n} \log^k n$ ($k = 1$ *or* 2), where successive terms arise from successive levels of recursion.

The analysis for the hypercube could only apply to the *PSC* computer if it is made to have some identical properties. If the recursive construction of a new network with a *PSC* nature is considered (this new network could be called the modified or augmented perfect shuffle computer (*MPSC*)), it becomes clear that within such a network, the same reasoning applies as for the *CCC* network. A *MPSC* with $2n$ PE's is recursively constructed by properly linking (according to the definitions given in Chapter 1) two *PSC*'s of size n. Figures 4.2.*a* and 4.2.*b* show the difference (bold lines) in terms of connections between a *PSC* and a *MPSC* each with 16 PE's.

Summarising the results obtained so far for the MCC^2 we can state that : /it Any $O(\log n)$ P-RAM algorithm based on the compression technique and in which the number of active PE's at the i^{th} iteration are the first $N/2^i$ numbered PE's, can be implemented in $O(\sqrt{n})$ parallel time on an $O(\sqrt{n} \times \sqrt{n})$ MCC^2 with shuffled row major indexing.

4.3 Generalisations

Our aim here is to expand the results obtained above. It will be shown that many problems of the general description given to the problem of evaluating a polynomial (i.e in terms of the parameters m, b and c) can, by observing certain strategic details, be solved in $O(\sqrt{n})$ parallel time on a MCC^2 and $O(\log n)$ (deterministically) on a CCC, PSC or $MPSC$.

The technique employed when solving the first problem (on the MCC^2, the CCC, the PSC and the $MPSC$) relied essentially upon reducing (by a constant factor) the size of the architecture occupied by the problem (during each iteration of a repeat statement). By doing so we were able on some architectures to reduce the costs of the recursive calls to the routing algorithms.

If at an arbitrary time the problem size is s, then the recurrence relation for the parallel computation time $T(s)$ is given by (1), where $f(s) = \sqrt{s}$ for the MCC^2 and $f(s) = \log^k s$ ($k = 1$ for probabilistic routing or $k = 2$ for deterministic routing) for the CCC, PSC and $MPSC$.

$$T(s) = T(s/2) + O(f(s)) \qquad s > 1$$

/bf (1)

$$T(s)=0 \qquad s=1$$

The solution to (1) is naturally $T(s) = O(\sqrt{n})$ for MCC's and $O(\log^{k+1} n)$ for CCC's, PSC's and $MPSC$'s.

For polynomial evaluation the confinement of successively produced problems into reduced portions of the MCC^2 was made possible by the use of the shuffled row major indexing of the PE's and it took, along with new problem creation, $O(1)$ time. On architectures such as the CCC or the $MPSC$, this was due to the natural mapping $a_i \rightarrow \text{PE}(i)$ but resulting in higher complexities.

Other problems identical to polynomial evaluation (for which $m = 1$, $b \leq 2$, $c = 0$ and where new problem creation takes $O(1)$ time) are those where the block $\{instructions\}$ inside the repeat statement defines an assignment of the type $a_j \leftarrow a_k \diamond a_l$, where $\diamond$ is a binary operator and where at each iteration the PE's with the j's indices are scattered around the architectures. For these type of problems to verify (1), it is necessary before each iteration to confine the new problem instance to a reduced area of the architecture.

This subclass could be divided into two categories. The first category consists of those where the order of choosing pairs of elements is not important. For instance, consider the problem of computing the sum of n elements. Due to the algebraic properties of addition such as distributivity, any way of choosing pairs would give the correct result. For this category of problems, confining a new instance of the problem to a reduced area (of successively indexed PE's) of the architecture would consist of sorting the contents of the active PE's after each iteration. In this way, the cost of invoking a library routing procedure can be reduced. Using the sorting algorithms of Nassimi and Sahni [NS79] or Thompson and Kung [TK77] will insure on the MCC^2 that the successive stages of confining recursively produced problems to reduced areas of the architecture are achieved in $\sum_{i=1}^{\log n/2} O(\sqrt{n}/2^{i-1}) + \sum_{i=1}^{\log n/2} O(\sqrt{n}/2^{i-1}) = O(\sqrt{n})$ time.

The phase of confining a new instance of the problem to a reduced area of the architecture can also be achieved using procedures of the same kind as *rank* and *concentrate* (chapter 3) used in the algorithm of Nassimi and Sahni [NS81]. Procedure *rank* can be used to assign a rank $r(\text{PE}(i))$ to every active $\text{PE}(i)$ such that $rank(\text{PE}(i) < rank(\text{PE}(j))$ if $i < j$ and $\text{PE}(i)$ and $\text{PE}(j)$ are both active. Procedure *concentrate* can used to move the contents of a $PE(i)$ to $\text{PE}(rank(\text{PE}(i)))$ which ensures the same result as that of sorting.

Figure 4.3 illustrates the results of the 'rank' and 'concentrate' procedures on a (4×4) MCC^2 with shuffled row major indexing.

The second category of problems consists of those that can be regarded as solvable (on the P-RAM model) using the tree contraction technique. In these problems, at each iteration, particular pairs of the elements acted upon have to be chosen carefully. Generally, these particular pairs are identified by pointers and therefore confining a new instance of the problem to a reduced area of the architecture cannot simply be achieved by sorting or the use of procedures such as *rank* and *concentrate* but will require other activities. The expression evaluation problem belongs to such a category. In section 4.6 we show what type of activities are necessary for an efficient implementation on the MCC^2 of the solution to this important problem.

However, for the problems examined above the phase of confining a problem to a reduced area is only necessary after each second iteration at which time the size of problem is reduced by a 4 factor and the cost of invoking a routing procedure is halved. This makes the overall time complexity of these stages $= \sum_{i=1}^{\log n} O(\sqrt{n}/2^i) = O(\sqrt{n})$. As a result, the recurrence relation for the parallel complexity time $T(s)$ for these problems on the MCC^2 will be given by (2).

$$\begin{aligned} T(s) &= T(s/4) + O(\sqrt{n}) && s > 1 \\ T(s) &= 0 && s = 1 \end{aligned} \qquad \textbf{(2)}$$

Referring to problems which are solvable on the P-RAM model using the techniques of compression or tree contraction as problems whose solutions are based on the compression of an input, we can conclude that :

Any $O(\log n)$ P-RAM algorithm based on the recursive compression of an input of length n can be implemented in $O(\sqrt{n})$ time on a $(\sqrt{n} \times \sqrt{n})$ MCC^2 with shuffled row major indexing provided that the successive confinements of recursively produced problems to areas where the cost of routing is reduced (by a constant factor) can be done in $O(\sqrt{n})$ time.

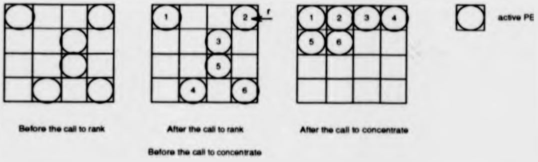


Figure 4.3: The result of procedures 'rank' and 'concentrate'

Without any further work, the same reasoning on the activities engaged on the MCC^2 in each of the cases applies to the CCC, PSC (and $MPSC$) except that the resulting time complexities are again not efficient (they are nearly efficient). This is because the complexities of sorting and procedures *rank* and *concentrate* are $O(\log^2 n)$ for a problem of size n [NS81].

However, there are cases where this can be improved. If $f(s)$ is a constant, then in (1), $T(s)$ is $O(\log n)$. This is possible if there exists a mapping of the graphical representation of our problems such that executing the instructions inside the repeat statement (of the type $a_j \leftarrow a_k \circ a_l$) is translated by invoking at each iteration some routing scheme taking a constant number of steps (independently of the size of the problem). It is easy to see that $T(s) = O(\log n)$ for problems where we can map (store) a_j to a PE which is directly linked to the PE's storing a_k and a_l. As for the MCC^2 we can state that : *Any $O(\log n)$ P-RAM algorithm based on the compression of an input of length n can be implemented on a CCC or PSC in $O(\log n)$ time provided that the graphical representation of the problem can be mapped on these architectures such that routing at any iteration of the* *algorithm is achieved in $O(1)$ time.*

4.4 Enlarging the class of problems efficiently solvable on feasible SIMD computers

The subclass of problems of R to be considered in this section are those that can be regarded as solvable using the 'divide and conquer' technique (with no combining stage) and therefore have $m > 1$. For the MCC^2, it is trivial to see that for any problem of size s which at each level of recursion is divided into 4 (or less) similar problems each of size at most $s/4$ will satisfy (2) provided that the problem division and the resulting assignment (confinement) of each created problem to its independent own quarter of the PE's can be achieved in $O(\sqrt{n})$. For the case of compression particular methods were used to confine a problem to reduced areas of the architecture. In the present case (where $m > 1$, $b \leq 4$), the assignment of each of the created subproblems can be done by means of some connected components algorithm, followed by sorting. The connected components algorithm will identify every subproblem by assigning a unique value to the nodes of its graphical representation. Sorting on these values will send each problem to a different area of contiguous locations of the architectures. On an MCC^2 the algorithm of Nassimi and Sahni [NS80] is the best known algorithm to achieve this purpose. Unfortunately this algorithm runs in $O(\sqrt{n} \times \log n)$ parallel time for graphs of arbitrary but fixed maximum degree d. However an exception that makes (2) hold is when $d = 2$ implying that in the general case other activities are necessary to convert the degrees of the graphical representation of the subproblems. Thus, using the best known connected components algorithm on an MCC^3 brings no improvement unless $d = 2$, but for any problem there may exist a way for avoiding the call to a connected components algorithm. For instance consider a problem (of size n) having a complete binary tree structure (where $d = 3$) in which at any iteration we can convert finding the global solution into that of finding the solutions to two subproblems associated with the right and left halves of the tree. To allocate each quarter of the tree to its portion of PE's (after each second iteration), we can for instance allocate a common value to each quarter of the leaves of the tree and communicate this same value to the internal nodes spanning those leaves. By simply sorting on the value acquired by each quarter of the tree we can achieve the confinement of the subproblems to reduced areas of the architecture without a connected components algorithm. Provided that the problem division for this problem and the transmission of values from the leaves to the internal nodes have overall $O(\sqrt{n})$ time complexity then the process described will also take $O(\sqrt{n})$ parallel time.

Finding the connected components of a graph on a CCC or PSC ($MPSC$) can be achieved respectively in $O(\log^2 n)$ (see [A89]) and $O(log^3 n)$ (see [NM82]) regardless of the degrees of vertices of the problem graph. Thus, the complexity bounds for solving the problems in hand on the CCC PSC ($MPSC$) become dominated by the complexity of the algorithm to find the connected components (if used). That is, for problems which have $m > 1$, the recurrence relation giving their time complexity is given by 3 where $p = 2$ for the CCC and $p = 3$ for the

PSC and *MPSC*

$$T(s) = T(s/2) + O(\log^p n) \qquad s > 1$$

$$\text{T(s)=0} \qquad \text{s=1} \qquad \textbf{(3)}$$

The solution to (3) is naturally $O(\log^3 n)$ (which is nearly efficient) for the *CCC* and $O(\log^4 n)$ for the *PSC* and *MPSC*.

A further generalisation is to consider replacing the factor 1/2 in (1) by $1/b$ where $b \geq 2$ is a constant factor. In this case, at each level of the recursion, a problem of size s is replaced with (up to) b similar problems each of size at most s/b (for simplicity, we take $s = b^j$ for some integer j). The initial difficulty faced is how to partition the PE's of our SIMD architecture so that each recursively created problem occupies an adequate set of successively indexed PE's. This set of PE's of which the size is determined by the parameter b will have to form a square on an MCC^2, belong to the same (sub)hypercube on a *CCC* or simply a set of adjacent PE's on the *MPSC*.

On the MCC^2, the shuffled row major indexing scheme used so far becomes unsuitable for odd values of the parameter b as it is based on the recursive division of the PE's into 2×2 blocks. Thus instead, a new indexing scheme: the *modified shuffled row major* indexing is used which recursively divides the PE's of our $(\sqrt{s} \times \sqrt{s})$ MCC^2 into a $b \times b$ matrix, where each sub square of the MCC^2 occupying an area of s/b^2 successively indexed PE's. Each sub square may then be occupied by one of the problems produced after a double application of the problem division procedure. Figure 4.4(c) shows the indices of the PE's of an MCC^2 indexed according to the *modified shuffled row major* indexing scheme when $b = 3$.

This new indexing scheme could be thought of as the result of displacing (re-indexing) blocks of b PE's originally with a row major indexing (figure 4.4(a)). To obtain a modified shuffled row major indexing scheme from a row major indexing scheme, we first divide the $b^2 \times b^2$ PE's into b vertical blocks and b horizontal blocks. The result is b^2 blocks of b^2 PE's. To achieve movements of blocks such as indicated in figure 4.4(b), every PE must know in which block it lies. Firstly, every PE knowing its index r and b^2, can compute its geometric coordinates (i, j), where $i = r \, mod \, b^2$ and $j = r \, div \, b^2$. Then every PE(i, j) computes the four parameters $B_1(i, j)$, $B_2(i, j)$, $B_3(i, j)$ and $rank(i, j)$. $B_1(i, j)$ and $B_2(i, j)$ indicate the position of the block of b^2 PE's in which PE(i, j) lies, $B_3(i, j)$ indicates in which (B_1, B_2) block of b PE's it lies and $rank(i, j)$ indicates its position inside the (B_3) block. These parameters are found as follows : $B_1(i, j) = i \, div \, b$, $B_2(i, j) = j \, div \, b$, $B_3(i, j) = (r \, div \, b^2) \, mod \, b$, $rank(i, j) = r \, mod \, b$.

Expressing r in terms of i and j, we obtain $r = jb^2 + i$. Furthermore i and j can be expressed as follows : $i = bB_1 + rank$, $j = bB_2 + B_3$.

Thus $r = (bB_2 + B_3)b^2 + bB_1 + rank$. The new indexing ($r_{new}$) for every PE($r$) is then obtained by interchanging the values B_1 and B_3. Thus $r_{new} = (bB_2 + B_1)b^2 + bB_3 + rank$.

On a CCC one application of the problem division procedure should send

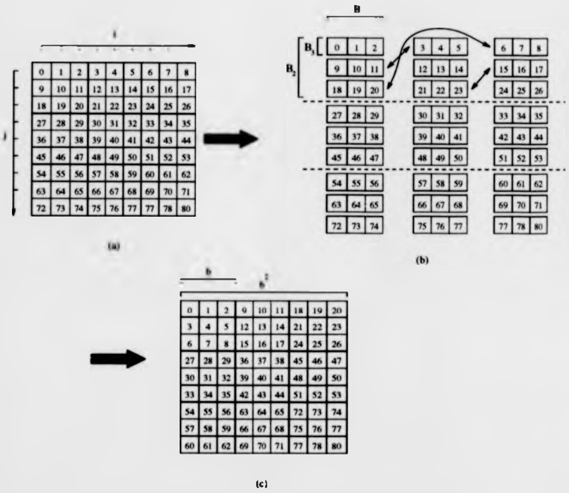


Figure 4.4: A modified shuffled row major indexing for $b = 3$

every set of s/b nodes of our problem to a set of PE's lying in the same dimension (that is if we want to stay within the same framework as for the MCC^2). Here the difficulty is that unlike the MCC^2 architecture of which the number of PE's is taken to be any natural number with a natural square root, the CCC has a strictly even number of PE's and makes it difficult to apply the the same strategy for all values of b. The same observation could be applied to the PSC and ($MPSC$).

Taking into account the constant c, and thus, considering the general case (where each problem of size s is recursively replaced by at most b similar problems each of size $\lceil s/b \rceil + c$, where b and c are constant integers, $b \leq 2$ and $c \leq 1$), we describe the implementation of the simple but archetypal list ranking problem.

4.4.1 Solving the list ranking problem

Given a list of elements, the list ranking problem is to associate with each element i of the list a parameter $dist(i)$, where $dist(i)$ is the distance from i to the head of the list. On a P-RAM, this problem can be solved in $O(\log n)$ parallel time by the use of the doubling technique. Initially, $dist(i)$ contains the (unit) distance between element i and the adjacent element which $P(i)$ points to. At the start of the computation the situation is illustrated (for a list of seven elements) at the top of figure 4.5.

After successive iterations and provided that $P(i)$ does not point to the head of the list, then the PE associated with i will execute $dist(i) \leftarrow dist(i) + dist(P(i))$ and $P(i) \leftarrow P(P(i))$. After k iterations we have P(i) (unless it points to the head

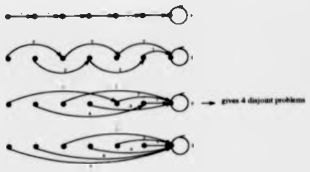


Figure 4.5: Ranking of a list of 7 elements

of the list) pointing 2^k elements along the list from i. Therefore, after $\lceil \log n \rceil$ iterations all P(i) point to the head of the list, placing the problem in the class NC [GR88].

If j is the head element of the list, then after each iteration within the algorithm outlined all the P(i)'s except P(j) form a number of directed paths terminating at j . If we observe that after each doubling operation the number of these paths is at most doubled, then it will be natural to use this in a recursive solution. Each of the problems of size s at one level of the recursion is replaced by at most two problems of size at most $\lceil s/2 \rceil + 1$. We obtain strictly disjoint (independent) subproblems by just duplicating the element at the head of the original list into each newly created list. These subproblems will be then distinguished by the use of the appropriate connected components. It should be noted that in the case of the MCC^2, our problem graph will have a maximum degree $d = 2$, a requirement that will let us use the $(O\sqrt{n})$ version the connected components algorithm of [NS80].

Generally, it is clear that if an initial problem of size n is replaced at the first level of the recursion by up to b similar problems, then each will be of size $(\lceil n/b \rceil + c) \le (n/b + c + 1)$. This size will decrease to $n/b^2 + (c+1)(1+1/b)$ at the second level of recursion and such that (for each of the b^i problems):

$$size(i) \le n/b^i + (c+1)\sum_{j=0}^{i-1}(1/b^j) = n/b^i + k(1 - 1/b^i), \;\; (k = b(c+1)/(b-1)).$$

Below a certain level of the recursion this estimate does not reduce the (integral) problem size and a minimum is reached when $(n-k)/b^i \le 1/2$ at which point $i \approx \log_b 2(n-k)$ and $size_{min} \approx \lceil k \rceil$. This provides the value of i at which the recursion bottoms out and at which the residual problem is solvable in $O(1)$ time.

The implementation of our strategy for the case where the parameter $c = 0$ is achieved by storing each problem of size $size(i)$ (at the i^{th} level of recursion) over n/b^i PE's (on the MCC^2 this is done as defined by our modified shuffled row major indexing scheme when b is odd). However, for the case where $c \ne 0$, instead of storing a single node of the problem graph at each PE, we now store up to k nodes of this graph at each PE, i.e, each problem of size $psize(i)$ is stored over n/b^i PE's and it is an easy technical problem to store the additional $\le k$ nodes evenly over these PE's. Then, within each application of the problem division procedure each PE in parallel handles in a sequential fashion the nodes it stores. Since there are a constant number of nodes at each PE, the time complexity of executing the problem division procedure will be of the same order as if a single node of the problem were stored at each PE. This ensures that the cost of solving a problem with $c = 0$ is the same as that of solving a problem with $c \ne 0$.

4.5 Solving the dynamic expression evaluation problem

Dynamic expression evaluation is the problem of evaluating an expression with no free preprocessing. This problem has seen solutions (based on the tree contraction technique) on the P-RAM model of computation by Miller and Reif [MR85] and Gibbons and Rytter [GR89]. The algorithm of [GR89] can be made to run in $O(\log n)$ parallel time using only $O(n/\log n)$ PE's. The version outlined and implemented in this section runs within the same time complexity but using $O(n)$ PE's. Such a version can be efficiently implemented on the MCC^2 and in near efficient manner on the CCC, PSC and $MPSC$.

The input to the algorithm of Gibbons and Rytter [GR88] is the expression tree, and the first step is to rank the leaves of the tree from left to right. The algorithm now repeatedly applies a so called *leaves cutting* operation that consists of reducing at each step the number of leaves of the expression tree by a factor of 1/2. At the end of the computation, the tree is reduced to a single node, at which time, the expression has been evaluated. The leaves cutting operation consists of the parallel removal of some leaves of the tree and is best introduced by describing how a single leaf may be removed (cut) by a local reconstruction of the expression tree. Figure 4.6 shows a portion of the tree before and after the removal of the leaf w_2, where $\circ$ is an operator and $f_i(x)$ is a function associated with the internal node w_i which when evaluated at $x =$ (value of the sub-expression associated with the subtree rooted at w_i) is the value to be passed to father of w_i.

Without going into specific details about local computations performed during

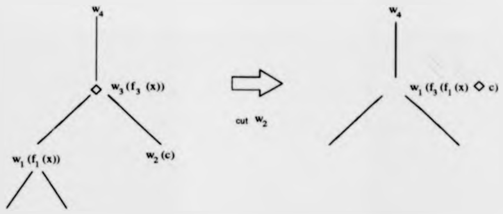


Figure 4.6: Illustration of the leaves cutting operation

the expression tree reconstruction which consist essentially of a constant number of pointer updates (The reader is referred to [GR88] and [GR89] for such clarifications), the whole algorithm consists of applying $\log n$ times the following operations that define the parallel leaves cutting operation :

1. in parallel cut all odd numbered leaves which are left sons
2. in parallel cut all odd numbered leaves which are right sons
3. in parallel divide the 'rank' of each leaf by two.

Figure 4.7 shows the expression tree for a given expression and Figure 4.8 shows the resulted trees after applying steps 1 and 2.

We discuss the implementation of the expression evaluation algorithm on the MCC^2 with its PE's indexed according to the shuffled row major indexing scheme.

An expression tree is a tree with $n-1$ nodes (for ease of description, we assume that n is a power of 2) where $(n/2)-1$ nodes are operators and $n/2$ nodes are operands. Before the construction of the expression tree (the expression is in the

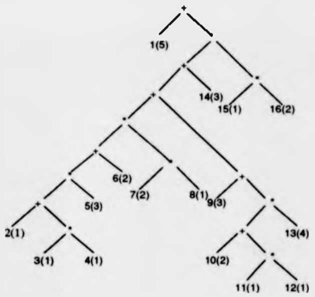


Figure 4.7

form of an array), operators can be recognised in $O(1)$ time and can be given numberings from 0 to $n/2 - 2$ using a subsequence ranking algorithm that runs in $O(\sqrt{n})$ (chapter 5). By the same procedure, the leaves can also be given numberings from $n/2$ to $n - 1$ and can also be ranked form 1 to $n/2$. The numberings obtained above will allow the mapping of the operators op_i (i = $[0, 1, 2, \ldots (n/2-2)]$) and the leaves $leaf_i$ ($i = [(n/2), (n/2+1), \ldots, (n-1)]$) onto the PE's of the $(\sqrt{n} \times \sqrt{n})$ MCC^2. When the tree is constructed every PE(i) will store a record consisting of either an operator op_i and two pointers or a leaf $leaf_i$ and a pointer corresponding to the adjacency list of the expression tree. Steps 1 and 2 of the leaves cutting operation are then executed respectively in $O(\sqrt{n})$ since they can be achieved by the use of a finite number of the operations of calls to the library routing procedure of Nassimi and Sahni [NS81] and step 3 takes only constant time. In the course of executing steps 1 and 2 the nodes which will not figure in the newly constructed tree (cut nodes) are marked as 'dead'. The tree that has to be constructed (input to the new iteration) will only contain 'live' nodes that were originally scattered over the PE's of the architecture. The problem now is how to achieve new problem creation. That is, how to make the new problem (set of live nodes) occupy a reduced portion of the architecture so that the pointers stay eligible.

We proceed by making each live node (having a record consisting of an *op*, and two pointers or a *leaf*, and a pointer) to record its address (labeled as old address) before the new problem creation phase. Then these live nodes are ranked and concentrated as already seen (thus will have new addresses). After this step every live node will write its new address into his old address. Finally every live node can update its pointers by reading the (new) information.

If the parallel leaves cutting operation is performed twice before each problem creation phase, then the newly created problem will occupy a square of size 1/4 the original area and the recurrence relation for the time complexity needed to evaluate an expression of size s will be given by(2), implying that expression evaluation can be achieved in efficient time on our the MCC^2. On the CCC and PSC computers our implementation will only lead to a nearly efficient solution, that is a parallel time $O(\log^{k+1} n)$ where $k = 1$ if probabilistic routing is used and $k = 2$ if deterministic routing is used.

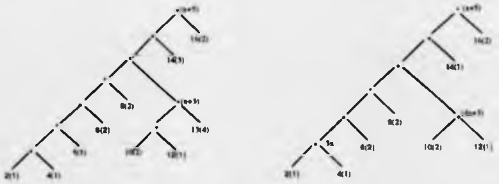


Figure 4.8:A leaves cutting operation performed on the tree of figure 4.7

4.6 Improving the processor utilisation

Many of the solutions to the problems treated in the previous sections had a poor processor utilisation. Processor utilisation (PU) is the average of the ratios of the number of PE's used at each step to the number of available PE's. For instance problems (of size n) with $m = 1$ have :

$$PU = (n + n/2 + n/4 + n/8 + \ldots + 1)/n \log n = 2/\log n$$

Barnard and Skillicorn [BS90] have suggested a method for increasing the PU on the CCC or hypercube by pipelining many identical algorithms. A simple illustration is the problem of computing the sum of kN numbers on N PE's. With the condition that one PE can only hold one data item this type of computation is broken down to k computations of the same type which is equivalent to k algorithms performing identical jobs. The strategy is to load a sequence of N elements, perform one required computation that makes half the PE's idle then to suspend this computation and input another sequence that will use the idle PE's and so on. At one stage all the PE's are non-idle except one. At any non loading stage of this pipelining strategy the same operation is performed on different instances (SIMD case) residing on different portions of the architecture. These portions of the architecture must be non-intersecting so as to avoid collision. On the hypercube with $N = 2^d$ PE's the last loaded sequence uses PE's forming a $(d-1)$ hypercube, the one loaded before last uses PE's in a $(d-2)$ hypercube and so on. Barnard and Skillicorn [B90] were able to find an algorithmic description for the non-intersecting sets of processors allocated to any instance of the pipelined computation at any stage. This was done by using orthogonal hyperplanes and rotating them around a diagonal of the hypercube. Their description also guarantees that a set PE's used by any instance at stage t is contained in the set of PE's used in stage $t-1$ so as to avoid unnecessary data movements.

To use an identical strategy on the MCC^2 would require finding non-intersecting planes and rotate them around the centre of the MCC^2. Unfortunately rotating planes on the MCC^2 (as shown in figure 4.9) cannot be done in a straightforward manner. Suppose we load an algorithm A and perform one computation that will make $n/2$ PE's idle. If we halt A and input another algorithm B, (B is a copy of A) then it is possible to find $n/2$ PE's to perform the computation of B (plane B in figure 4.9) and $n/4$ PE's to perform the computation of A (plane A). Furthermore plane B is obtained from rotating plane A around the centre of the MCC^2. If we halt algorithms A and B and input another algorithm C (C is identical to A and B), then it will still be possible to find three non intersecting sets of PE's to perform the required computations. The PE's used by algorithm C are obtained by rotating plane B and those used by B are obtained by rotating plane A. Now

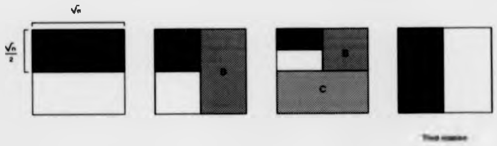


Figure 4.7: Rotating $n/2$ PE's in a mesh

to input another algorithm D and find 4 non-intersecting (planes) sets of PE's cannot be done without moving data around (i.e displacing planes). Therefore to input the 4th algorithm D we will have to wait another $\log n - 2$ iterations for A to finish. This implies that following the method of rotating planes on the MCC^2 we can only have at most 3 algorithms (performing the same operations) at any one time. This is not as good as on the hypercube but will nevertheless increase the PU parameter for many computations.

To allocate the PE's following the above method, we proceed by assigning to every algorithm a rank m ($m = 0\, to\, N - 1$) indicating its position in a queue of N identical algorithms to be input (pipelined). Following the observation made above, we can determine at which time step the algorithm with rank m will be loaded. On the MCC^2 between time $t = 1$ and time $t = \log n$ we can only load the algorithms with ranks 0, 1 and 2. Algorithm with rank 0 is input at $t = 0$, algorithm with rank 1 is input at $t = 2$ and algorithm with rank 2 is input at time $t = 4$. The algorithms with ranks 3, 4 and 5 are input between $t = \log n + 3$ and $\log n + 7$. Following the same reasoning we can deduce that every algorithm with rank m is input at time:

$$t = (m\, div\, 3) \log n + m \bmod 3 + m$$

Having determined the loading time for every algorithm, we can now provide a description of the sets of PE's that will be used by any algorithm at any stage. To simplify this description we define the operations Compress Up, Compress down, Compress Right and Compress Left. Let PE(i,j)'s $(i = a_0, a_1, a_2, \ldots a_{n-1})$, $(j = b_0, b_1, b_2, \ldots b_{n-1})$ be the PE's used by an algorithm at stage $t-1$. The operation Compress up will cause an algorithm to use at step t the PE(i,j)'s $(i = a_0, a_1, a_2, \ldots a_{n-1})$, $(j = b_0, b_1, b_2, \ldots b_{n/2-1})$. The operation Compress down will cause the use of PE(i,j)'s $(i = a_0, a_1, a_2, \ldots a_{n-1})$, $(j = b_{n/2}, b_{n/2+1} \ldots b_{n-1})$. The operation Compress left will cause the use of PE(i,j)'s $(i = a_0, a_1, a_2, \ldots a_{n/2-1})$,$(j = b_0, b_1, b_2, \ldots b_{n-1})$ and operation Compress Right will cause the use of PE(i,j)'s $(i = a_{n/2}, a_{n/2+1}, \ldots a_{n/2-1})$, $(j = b_0, b_1, b_2, \ldots b_{n-1})$. The rank m of every algorithm will determine what type of operations are to be performed on it. If we choose to rotate planes in a clockwise fashion then after the loading step we allocate the upper half of the PE's to the first loaded ($m = 1$) algorithm (A in figure 4.10), the right half to the second loaded B ($m = 2$, the lower half to the third and the left half to the fourth. The same will happen to the next four algorithms and so on. Our description is complete if we compute for every algorithm the parameter $type = m \bmod 4$ and at the time it is loaded we set a step counter to 0. Knowing its loading time and according to the value of *type* the PE's used by any algorithm at any step are described by:

For algorithms of *type* = 0

Loading step : Allocate all the Processors

Odd step : Compress Up

Even step : Compress Left

For algorithms of *type* = 1

Loading step : Allocate all the Processors

Odd step : Compress Right

Even step : Compress Up

For algorithms of *type* = 2

Loading step : Allocate all the Processors

Odd step : Compress Down

Even step : Compress Right

For algorithms *type* = 3

Loading step : Allocate all the Processors

Odd step : Compress Down

Even step : Compress Right

Using such a pipelining strategy increases the computation time of an algorithm from $O(\sqrt{n})$ to just $O(\sqrt{n}) + c$ ($c \leq 2$ represents the number of times an algorithm is halted to load another algorithm) but surely does increase the overall processor utilisation. To compute the new processor utilisation it is enough to consider the computation between times $t_1 = \log n + 2$ and $t_2 = 2\log n + 5$ where t_1 is the time at which occurs the first computation after the system has already

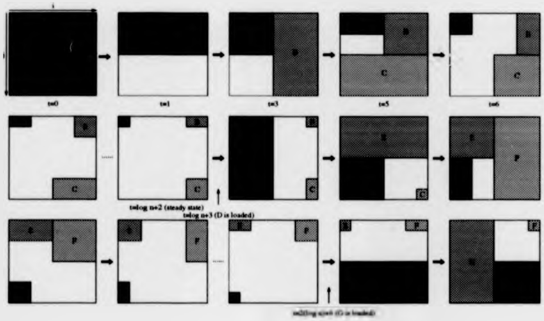


Figure 4.8: Pipelining algorithms on the MCC^2

reached a steady state (i.e $\forall t \geq t_1$ there is always three active algorithms), and t_2 is the step time before the system cycles again. Moreover if only the non loading steps are considered the PU is computed as follows. At time $t = t_1$ an algorithm loaded at $t = t_1 - 1$ is using $n/2$ PE's and the two previous active algorithms are using just $2+1$ PE's. At $t = t+2$ the number of used PE's becomes $n/2+n/4+1$ (a second algorithm has been input) and at $t = t+4$ (a third algorithm has been input) the number of used PE's is $n/2+n/4+n/8$. Because no algorithm is again input up to $t = t_2 + 1$, the expression giving the PU is:

$$\approx \{(n/2+1+2)+(n/2+n/4+1)+(n/2+n/4+n/8)+(n/4+n/8+n/16)+\ldots$$

$$+\ldots+(1+2+4)\}/\log n$$

$$3 \times 2/\log n$$

This is an improvement over the previous PU.

Chapter 5

The balanced binary technique on feasible SIMD computers

5.1 Introduction

In the previous chapter, the general techniques of compression, tree contraction and 'divide and conquer' were shown to be of great facility for the design of efficient algorithms on the mesh for a wide class of problems. In this chapter we show that the general technique of the balanced binary tree (commonly employed in optimal P-RAM algorithms) may also lead to efficient problem solving on the mesh connected computer. In this respect other architectures are dealt with later. Some P-RAM algorithms employing this technique cannot be implemented on the mesh by simply using some embedding of a complete binary tree along with calls to a library routing algorithm. One reason, for example, is that concurrent reads in the P-RAM model (the number of which can be logarithmic in the input size) may lead to slower computation (i.e $O(\log n\sqrt{n})$). However, new and efficient algorithms avoiding such problems can often be devised which nevertheless use a related balanced binary tree approach. Such an example will be described in sections 5.4 and 5.5.

The particular examples of sections 5.4 and 5.5 are efficient algorithms for bracket matching on the mesh. That is, given a string of n brackets, the i^{th} bracket (for all i, $0 \leq i \leq n-1$) may learn the position (in the string), called match(i), of its matching bracket in $O(\sqrt{n})$ parallel time on a $(\sqrt{n} \times \sqrt{n})$ MCC^2. It follows that, given an arithmetic or algebraic expression presented as a string of symbols, the tree form of the expression can be constructed (by an easy extension to the bracket matching algorithm) with similar algorithmic efficiency (see [BV85], [GR88]). This extends a number of previous results. It was shown in the last chapter that, if an expression is presented as an expression tree, then the expression can be evaluated in $O(\sqrt{n})$ parallel time on a $(\sqrt{n} \times \sqrt{n})$ MCC^2.

For algebraic expressions, such an evaluation requires that the corresponding algebra has a carrier of constant-bounded size. The recognition of bracket and input driven languages can be reduced to the computation of such algebraic expressions (see [GR88]). It follows that if the input is in the form of a string (of the symbols making up the expression) stored in an array, the following problems have efficient solutions on the mesh :

(a) Evaluation of arithmetic expressions.

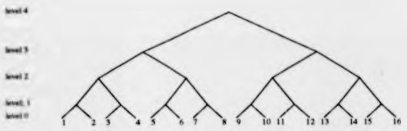


Figure 5.1

(b) Evaluation of algebraic expressions with carrier of constant-bounded size.

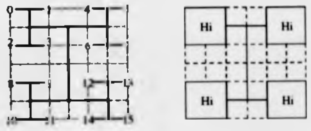
(c) Parsing expressions of both bracket and input driven languages.

As a by-product, two further (but comparatively trivial) problems, prefix sums and sub-sequence ranking, are shown to have efficient solutions in section 5.3. Indeed, this general technique is likely to yield efficient solutions for many more problems.

The balanced binary tree method (chapter 2) over a string of n characters $(c_0, c_1, \ldots, c_{n-1})$ employs a balanced binary tree with the characters placed at the leaves. Figure 5.1 shows such a tree for $n = 16$.

As indicated in chapter 2, some computations over such a tree might be simple and therefore will have a straightforward implementation. However for other problems (such as the bracket matching problem) this might not be the case and it would be necessary to devise some specific adaptations such as embedding the balanced binary tree.

A standard (explicit) way of embedding a balanced binary tree in the mesh is to use the so called H-tree representation (see for example [U84],[A89]). The



(a) H_4 (b) Construction of H_{i+1}

Figure 5.2

i^{th} H-tree, H_i, is an embedding of the balanced binary tree with 4^i leaves. Thus, figure 5.2(a) shows H_4 (the embedding of the tree of figure 5.1), the leaves of H_4 are numbered according to the left to right order of figure 5.1). Figure 5.2(b) indicates the inductive construction of H_{i+1} from H_i.

Although we have dilation $= O(\sqrt{n})$, expansion=1 and congestion=1, a drawback of this construction is that the balanced binary tree n leaves requires a large mesh of $(2\sqrt{n}-1) \times (2\sqrt{n}-1)$ PE's. However, it is possible to employ a strictly $(\sqrt{n} \times \sqrt{n})$ mesh for our purposes as described in the next section.

5.2 Implicit representation of the balanced binary tree

The H-tree embedding of balanced binary trees in the mesh requires more PE's than necessary. Some of the additional elements are used for the disjoint representation of all tree edges (i.e. routing paths on the mesh) and others are completely unused. It is not necessary for all such edges to be disjointly represented (i.e we do not need the congestion= 1), in fact (for our purposes) only those edges at the same height have to appear in disjoint regions of the mesh. We therefore adopt

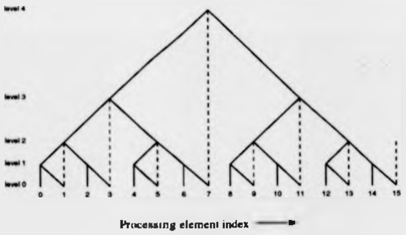


Figure 5.3

the approach of associating binary tree nodes with PE's indices (at this point we do not specify where each such indexed element is placed in the mesh) as indicated in figure 5.3. Two tree nodes (a leaf and an internal node) are associated with (or 'stored at') each PE. In figure 5.3, **vertical** lines (either solid or dashed) connect the two nodes which are associated with the PE whose index appears at the bottom of each vertical line.

Figures similar to figure 5.3 are inductively constructed as indicated in figure 5.4. This construction which associates tree nodes with PE's has the following properties:

(1) Consecutive tree nodes at level j which are both left (or right) sons are stored at PE's whose indices differ by 2^{j+1} .

(2) At level j, the first non leaf node which is a left son occurs at PE index $(2^{j-1} - 1)$ and the first non-leaf node which is a right son occurs at index $(3 \times 2^{j-1} - 1)$.

(3) Every PE with an even (respectively, odd) index stores a leaf which is a left (right) son.

Thus, if every PE knows its own index i and the number of leaves n, and if every PE executes the following instructions:

```
$k \leftarrow 2$, level$\leftarrow$ 'none', typeofson$\leftarrow$ 'none'
for $j = 1$ step 1 until $\log n$ do
    begin
        $k \leftarrow 2k$
        if remainder $((i+1)/k) = k/4$ then
            begin
                $level \leftarrow j$, $typeofson \leftarrow left$
            end
        if remainder$((i+1)/k) = 3k/4$ then
            begin
                $level \leftarrow j$, $typeofson \leftarrow right$
            end
    end
```

Then after $O(\log n)$-time (and because of properties (1) and (2)), every PE knows the type of non-leaf tree node left or right) associated with it and at what level in the tree this node is. Additionally each PE may determine whether it is associated with the root by checking if $j = \log n$. Similarly (employing property

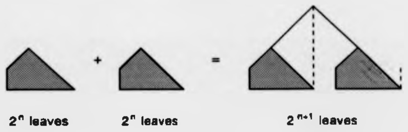


Figure 5.4

(3)), each PE may easily determine in $O(\log n)$ time if the leaf associated with it is a left or right son.

Consider now the distribution of PE's across the MCC^2. We use the shuffled row major indexing scheme which is important to establish the lemma stated at the end of this section. With this indexing the binary tree construction has the property that son to to father (and father to son) routing can be performed on the basis of local information only. Figure 5.5 shows how each PE (knowing the level = j and type of an associated tree node) knows the route to the next PE (associated with the father node). For example, a PE associated with a tree node which is a left son at level j ($j > 0$ and odd) will find the processing element storing the father of this node at a distance of $2^{(j-1)/2}$ mesh steps to the right.

Figure 5.6 shows (through (a) to (d)) routing from the leaves to the root for the balanced binary tree with 16 leaves.

lemma 5.1 For a complete binary tree with n leaves, the total time to route (in parallel) messages of constant bounded length from leaves to root (or vice versa) of the tree can be achieved in $O(\sqrt{n})$ parallel time on $(\sqrt{n} \times \sqrt{n})$ MCC^2.

Son to father routing
(distances in mesh steps)

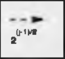
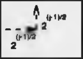
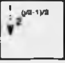
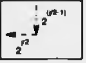
level of son	left son	right son
j=0	∅	←
j>0 j odd	 $2^{(j-1)/2}$	 $2^{(j-1)/2}$ $2^{(j-1)/2}$
j>0 j even	 $2^{(j/2-1)/2}$	 $2^{(j/2-1)}$ $2^{j/2}$

Figure 5.5

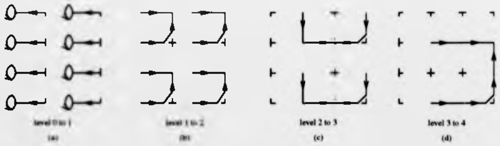


Figure 5.6

Proof It is sufficient to consider only routing from the leaves to the root. Figure 5.5 shows that the maximum length path (in terms of mesh steps) from a leaf to the root as traced on the mesh is that corresponding to repeated right-son to father routing. A section from such a path in moving from an odd level $j > 0$ in the tree to the next odd level $j + 2$, has length (in mesh steps):

$$(2^{(j-1)/2} + 2^{(j-1)/2}) + (2^{(j+1)/2-1} + 2^{(j+1)/2}) = 5 \times 2^{(j-1)/2}$$

In going from level 0 to level 1, such a path uses one mesh step. Thus if the root (at level $\log n$) is at an odd level, the path has overall length:

$$(1 + \sum_{j=0}^{((\log n)-2)/2} 5 \times 2^j) = 5\sqrt{n/2} - 4$$

mesh steps.

On the other hand, if the root is at an even level, then the path has overall length:

$$(1 + (\sum_{j=0}^{((\log n)-3)/2} 5 \times 2^j) + \sqrt{n}) = 7\sqrt{n}/2 - 4$$

mesh steps, where on the left hand side of the equation, the term $\sqrt{n}$ is the length of the final section of the path from level $(\log n) - 1$ to the root and we immediately state the following corollary:

Corollary 1. Any P-RAM algorithm based on the balanced binary tree will have (within a constant multiplier) an efficient implementation on the MCC^2 (That is a parallel computation time of $O(\sqrt{n})$ for inputs of size n) provided that :

(a) Its parallel updown activity on the binary tree is time-bounded by a constant number of leaf to root (or root to leaf) routings

(b) All operations at nodes are performed in constant-bounded time

(c) It sends father to son (and son to father) messages of constant bounded length only.

5.3 Elementary examples

In this section we describe the problems of partial sums computations (already seen in chapter 3) and subsequence ranking which satisfy corollary 1. These algorithms appear as sub-tasks in the algorithm of the next section. Moreover, at the end of this section we describe how the tree nodes of the balanced binary tree may be pre-order numbered by an efficient algorithm.

5.3.1 Partial sums computation

Given n values $(v(0), v(1), \ldots, v(n-1))$, a partial sums computation evaluates, for all i, $(0 \leq i \leq n-1)$, each of the sums $\sum_{j=0}^{i} v(j)$. The computation starts with (for all s, $0 \leq s \leq n-1$) $v(s)$ being stored at the PE associated with leaf s (the leaves are numbered from left to right). During a computation a PE associated with tree node i uses three storage locations A_i, B_i and C_i. At the outset of the computation, (for all s, $0 \leq s \leq n-1$) $v(s)$ at the s^{th} leaf is assigned to A_j. Then at successively higher levels in the tree, all non-leaf nodes (those at the same level in parallel) compute:

$$A_i \leftarrow A_{leftson(i)} + A_{rightson(i)}$$
$$B_i \leftarrow A_{rightson(i)}$$

After the root has computed these values it sets $C_{root} = A_{root}$ and then non-root nodes at successively lower levels in the tree (those at the same level in parallel) compute:

$$C_{i(is\,a\,left\,son)} \leftarrow C_{father(i)} - B_{father(i)}$$
$$C_{i(is\,a\,right\,son)} \leftarrow C_{father(i)}$$

An invariant of the computation is that if tree node i is the root of a subtree spanning leaves r to s then C_i equals $\sum_{j=1}^{s}$ value(j). Thus when the computation stops, for each leaf i, C_i is the partial sum $\sum_{j=1}^{i}$ value(j).

5.3.2 Subsequence ranking

Given a string (e.g. YABBABBAXABABA), the subsequence ranking problem is to rank the items in a sublist of distinguished items (e.g. the B's). The characters of the string are placed at the leaves of the tree. Each PE (storing a leaf) has a memory location which is initially made to contain 1 if the associated character is distinguished (is a B in the example) otherwise it contains 0. This is schematically shown in (i) below. A prefix computation is then performed on these values (the result for such a computation for our example is shown in (*ii*) below). For each distinguished item, the associated storage location then contains its ranking (the contents of such storage locations can be locally nulled for non-distinguished items as (*iii*) below illustrates).

String:

$$YABBABBAXABABA$$

(i) assign values

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

(ii) perform a prefix computation

0 0 1 2 2 3 4 4 4 4 5 5 6 6

(iii) zero non-distinguished items

0 0 1 2 0 3 4 0 0 0 5 0 6 0

Sometimes it is useful (and this is true for the example of next section) for each each tree node to have a unique defining integer (perhaps its preorder number). The following algorithm determines a preorder numbering of the tree nodes. Each PE associated with a tree node i has two storage locations (registers) A_i and B_i. Initially, for all leaves, $A_i \leftarrow 1$. Then the computation proceeds up the tree in the usual way with the non-leaf nodes computing:

$$A_i \leftarrow A_{leftson(i)} + A_{rightson(i)} + 1$$

In this way, A_i for all tree nodes becomes equal to the number of nodes in the subtree rooted at tree node i. When A_{root} has been computed, the assignment

$B_{root} \leftarrow 1$ is made and then the computation proceeds down the tree with non-root nodes computing:

$$B_{i(a\,left\,son)} \leftarrow B_{father(i)} + 1$$
$$B_{i(a\,right\,son)} \leftarrow B_{father(i)} + 1 + A_i$$

When the PE's associated with the leaf nodes have finished their computation, B_i for all nodes is the preorder number of that node.

5.4 Solving the bracket matching problem

Given a sequence of n brackets [the sequence (( ) ((( ))) (( )) ( )) is used as an example], the bracket matching problem is to compute the function *match*[i] which for all i, $0 \leq i \leq n-1$ is the position (in the string) of the bracket matching that at position i. For our example match[2]=1 match[3]=8. The P-RAM algorithm of Bar-On and Vishkin [BV85] is essentially an algorithm that computes the function *match*. Knowing match (as [BV85] indicates), it is an easy extension to compute (in constant time on a P-RAM) the expression tree from an expression presented as a string.

The algorithm described in [BV85] is not readily implemented on the mesh in time $O(\sqrt{n})$. This is because (c) of corollary 1 is not satisfied. Bar-On and Vishkin's algorithm consists mainly of an upward phase in the tree followed by a downward phase in the course of which each bracket follows its unique path in the tree of partial results from the leaf at which it is stored to the leaf storing

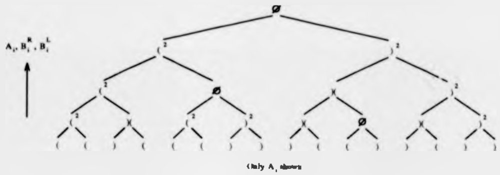


Figure 5.7

its matching bracket. Figure 5.7 illustrates such a tree for our example sequence. During the upward phase many paths intersect thus violating (c) of corollary 1. In our following algorithm, the upward phase of [BV85] is simulated by a downward (step 2), and the downward phase (of [BV85]) is replaced by a new technique contained in steps 3 and 4. This new algorithm satisfies corollary 1.

Given a sequence S of brackets, reduced[S] is the sequence obtained from S by repeatedly deleting adjacent pairs '( )' [GR88], e.g. reduced[))(())(] =))(. In general, any irreducible sequence of brackets is of the form $)^i(^j$. Therefore, a pair of integers are sufficient to represent any reduced form. Given any two reduced sequences $S_1 =)^i(^j$ and $S_2 =)^k(^l$, it is possible to compute reduced$[S_1S_2]$ in $O(1)$ time :

$$\mathit{reduced}[)^i(^j)^k(^l] \leftarrow \textbf{if } k \geq j \textbf{ then })^{i+k-j}(^l \textbf{ else })^i(^{l+j-k}$$

In order to compute the function *match*, we employ the balanced binary tree with the brackets placed at the leaves of the balanced binary tree. After the execution of step 1, A_i will store the reduced form of the sequence of brackets which are stored at the leaves of the subtree rooted at i. Moreover, B_i^R (respectively B_i^L) will store the reduced form of the sequence of brackets stored at the leaves of the tree rooted at the left (right) son of i. The superscript here refers to the direction in which the contents of the location B_i^R or (B_i^L) will be passed down the tree in step 2 of the algorithm and so is contrary to a seemingly natural superscripting at this point. Initially every PE storing a leaf of the tree, A_i is set equal to the type of bracket associated with that leaf.

Step 1. We start with the input sequence of brackets at the leaves of the balanced binary tree (for each leaf, A_i is the bracket stored at that leaf) and in an upward phase we compute, for all non-leaf nodes i :

$$B_i^R \leftarrow A_{leftson(i)},$$
$$B_i^L \leftarrow A_{rightson(i)}$$
$$A_i \leftarrow reduced[B_i^R B_i^L]$$

Figure 5.7 shows the result of applying this step for our example string.

Step 2 In this step, each non-leaf node i (in parallel) sends down the tree the value of B_i^R (respectively, B_i^L) to every node of the tree rooted at the right (left) son of i. When each of these values passes through a node, it is copied to both sons of that node. Thus each leaf receives a stream of B values (the first from its father, the next from its grandfather and so on). On receiving the current B value a computation taking $O(1)$ time is performed at each leaf. Internal nodes (in turn those at level 1, than those at level 2 and so on) send their values to the leaves. The parallel computation time for internal nodes covering k leaves is $O(\sqrt{k})$, and so overall $O(\sqrt{n})$ parallel time is required.

After this step of the algorithm, the i^{th} leaf (for all i, $0 \le i \le (n-1)$) will know the pre-order number of the least common ancestor of the leaves with indexes i and match(i). The least common ancestor of two leaves is that ancestor with the lowest level number. During the computation each non-root tree node i stores a triple T_i, initialised as follows :

```
$T_i(1) \leftarrow$ prefix number of tree node $i$
$T_i(2) \leftarrow$ if $i$ is a left son then $B^L_{father(i)}$
            else $B^R_{father(i)}$
$T_i(3) \leftarrow$ if $i$ is a left son then $L$ else $R$
```

In addition, if i is a leaf then there is a second triple L_i, initialised as follows :

$L_i(1) \leftarrow 0$

$L_i(2) \leftarrow \emptyset$

$L_i(3) \leftarrow$ **if** i stores a left bracket **then** L **else** R

The stream of B values that each leaf receives is in fact transmitted by sending down the tree the triples T_i. The additional information stored in these triples is employed to guide the computation that takes place at the leaves. If i is a leaf, then the arrival of each new T_i induces the following computations :

```
if $L_i(1) = 0$ then
  if $L_i(3) = T_i(3)$ then
    begin
          if $L_i(3) = L$ then
            begin
                  $L_i(2) \leftarrow$ reduced[$L_i(2)T_i(2)$]
                  if $L_i(2)$ begins with')' then
                    $L_i(1) \leftarrow T_i(1)$
            end
          else
            begin
                  $L_i(2) \leftarrow$ reduced[$T_i(2)L_i(2)$]
                  if $L_i(2)$ ends with '(' then
                    $L_i(1) \leftarrow T_i(1)$
            end
    end
```

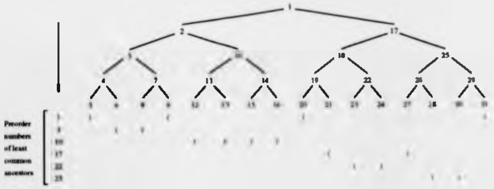


Figure 5.8

After all leaves have performed this computation for the last time, $L_i(1)$ stores the prefix number of the least common ancestor of leaf i and the leaf which stores the bracket matching the one at leaf i. For our example input sequence, the result of applying this step is shown in figure 5.8.

In order to understand how this step works, consider the case that a left bracket is stored at a particular leaf. For this leaf $L_i(3) = L$. Each time a value $T_i(2)$ arrives at the leaf, $L_i(2)$ which is initially the empty sequence, is updated as follows : $L_i(2) \leftarrow$reduced$[L_i(2)T_i(2)]$. When $L_i(2)$ begins with a right bracket, the least common ancestor is given by $L_i(1)$. The computation works (for left brackets at leaves) because of the following invariant. Suppose that i is the root of a subtree spanning leaves p to q and that the leaf is at position r (between p and q), then after the assignment : $L_i(2) \leftarrow$ reduced$[L_i(2)T_i(2)]$, $L_i(2)$ is the reduced form of the brackets stored from positions $(r+1)$ to q.

Step 3. Following step 2, the subsequence of brackets all having the same least common ancestor form a string of left brackets followed by the string of their right matching brackets :

$$(,(,\ldots,(,),),\ldots,).$$

For a given least common ancestor, we rank (by the previously described algorithm for subsequence ranking) the brackets to obtain :

$$(_1,(_2,\ldots,(_k,)_{k+1},)_{k+2},\ldots,)_{2k}$$

where subscripts denote the ranks of the brackets. However, the desired subscripting should be as follows :

$$(_1,(_2,\ldots,(_k,)_k,)_{k-1},\ldots,)_1$$

This is easily achieved by causing each PE storing a right bracket with rank r (and knowing k which is passed down the tree in the ranking computation) to compute the new subscript $(2k-r+1)$ in constant time. Now such a subscripting has to be obtained for all possible least common ancestors. Every non-leaf tree node is such a possible ancestor. The computations for all non-leaf nodes at the same level can be performed in parallel and if the subtrees rooted at a certain level have k leaves, all computations for this level will take $O(\sqrt{k})$ parallel time. Summing over all levels gives a computation time for this step of:

$$\sum_{i=0}^{\log n-1} \sqrt{n/2^i}$$

Step 4 At this stage every bracket stored at a leaf knows :

(a) The least common ancestor (A) shared with its matching bracket.

(b) Its subscript (S) from step 3.

(c) Whether it is a left (L) or right (R) bracket. Denote L or R by B.

This step then simply sorts in $O(\sqrt{n})$ parallel time all brackets according to the triple (A, S, B) associated with each bracket using the sorting algorithm of [NS79] or [TK77]. Let $L < R$ for sorting purposes. If '(' ends up at PE(i), then its matching bracket will be stored at PE($i + 1$). If in the sorting phase each bracket carries with it its initial address, then matching brackets can exchange addresses and then return to their original positions by resorting on their own addresses.

Summing up the time complexities for all phases, we obtain an $O(\sqrt{n})$ parallel time algorithm for the bracket matching problem. The $O(\sqrt{n})$ time complexity for most steps relied heavily on the routing schemes induced by the implicit embedding of the balanced binary tree. Our algorithm can also be implemented in time $O(\log n)$ on a CCC using complete binary tree embeddings on this architecture such that of Wu [W85] and Gibbons and Ravindran [GRa92]. In these embeddings routing form father to son or vice versa is achieved in constant time.

5.5 Another solution to the bracket matching problem

This section is at the crossroad between the last chapter and the last section. We describe a second algorithm that provides a recursive solution to the bracket matching problem.

For a given correct sequence of parenthesis we start by computing the so called tree of partial result. Then for each bracket stored at PE(i) a parameter c_i is computed which will indicate whether a bracket has its match lying in the same half as itself (such a bracket is called a matched or unmatched bracket) or not. The operations to be performed after this step are those of shifting specific subsets of unmatched brackets from the left half to the right half and vice versa to obtain two sequences (halves) where all the brackets and their matchings lie in the same (half) sequence. We then consider each half separately as an independent subproblem and recursively repeat the process. The algorithm terminates after $\log n$ iterations (for a sequence of length n), time when a bracket and its matching will be in contiguous locations. To understand how this algorithm runs, we illustrate the steps of its first iteration (on the MCC^2) on the following input sequence of 16 brackets:

$$(_1\,(_2\,(_3\,(_4\,)_5\,)_6\,(_7\,(_8\,)_9\,)_{10}\,(_{11}\,(_{12}\,)_{13}\,)_{14}\,)_{15}\,)_{16}$$

Subscripts indicate the positions of the brackets in the array. The computation of the tree of partial results is obtained as in the first step of the previous algorithm. Using this search tree and proceeding as in step 2 of the same algorithm it can be decided for every bracket if its matching lies in the same 'half or not. This is done by checking the level in the tree of its least common ancestor which its shares with its matching bracket. This information is contained in c_i. A bracket will know that its match lies in the same half if $c_i < \log n$.

The next sequence shows the set of brackets (indices in bold) that do not have their matching in the same half (unmatched brackets). The number (M) of such brackets is computed by assigning a value v_i to every bracket, where $v_i = 1$ for every matched one and $v_i = 0$ for the unmatched ones and then sort the records H_i's consisting of the bracket, its address (i) and its v_i value as follows: $H_i < H_j$ if $v_i < v_j$ or $v_i = v_j$ and $i < j$. We can then determine M by making every PE(i) compare v_i to v_{i+1}. The number M is equal to the index k of the PE(k) for which $v_k \neq v_{k+1}$. After this computation, the brackets are sent back to their original addresses. For our example $M = 8$.

$$(_{\mathbf{1}}\,(_{\mathbf{2}}\,(_{3}\,(_{4}\,)_{5}\,)_{6}\,(_{\mathbf{7}}\,(_{\mathbf{8}}\,-\,)_{\mathbf{9}}\,)_{\mathbf{10}}\,(_{11}\,(_{12}\,)_{13}\,)_{14}\,)_{\mathbf{15}}\,)_{\mathbf{16}}$$

It is obvious to see that in each half of the sequence $M/2$ (for our example 4) brackets are unmatched. Our strategy is to shift $M/4$ brackets from the left half to the right half and vice versa. so that the result is two correct subsequences. The $M/4$ unmatched brackets to be shifted are determined by the following observation: If we divide a correct sequence of brackets into two halves and after that we match in every half every bracket that can be matched, then the remaining brackets in the left half are all left brackets and the ones in the right half are all right brackets. Moreover, every k^{th} leftmost unmatched bracket in the left half must have as its matching the k^{th} rightmost unmatched bracket in the right half. Therefore the sets of unmatched brackets to be shifted from each half (to give two correct sequences) are those consisting of half the number of unmatched parenthesis in each half lying at the rightmost positions.

The shifting operation is achieved by first sorting in each half separately the records H_i's consisting of the bracket B_i, v_i and i according to : $H_i < H_j$ if $v_i < v_j$ or ($v_i = v_j$ and $i < j$). For our example, the result of such sorting follows:

$$(_1 (_2 (_7 (_8 (_3 (_4)_5)_6 \; - \;)_9)_{10})_{15})_{16} (_{11} (_{12})_{13})_{14}$$

The the shifting is simply achieved by the following procedure (where C is just an auxiliary register):

```
for all i, $M/4 + 1 \le i \le M/2$ in parallel do
    begin
    $C_i \leftarrow B_i$
    $B_i \leftarrow B_{n/2+i}$
    $B_{n/2+i} \leftarrow C_i$
    end
```

For our example, the brackets shifted are $(_7$, $(_8$ from the left half and $)_{15}$, $)_{16}$ from the right half and we obtain the following two sequences.

$$(_1 (_2)_{15})_{16} (_3 (_4)_5)_6 -)_9)_{10} (_7 (_8 (_{11} (_{12})_{13})_{14}$$

However, such a shifting causes that in the right half we have a set of left brackets standing on the right of their matching right brackets. For our example where $(_7$ and $(_8$ should be on the left of $)_9$ and $)_{10}$. Therefore, another shifting (correcting) operation is necessary for the second half. This is simply achieved by executing:

```
for all $i$, $1 \leq i \leq M/4$ in parallel do
    begin
    $C_i \leftarrow B_i$
    $B_i \leftarrow B_{M/4+i}$
    $B_{M/4+i} \leftarrow C_i$
    end
```

The corrected sequences for our example are:

$$(_1\,(_2\,)_{15}\,)_{16}\,(_3\,(_4\,)_5\,)_6\;-\;(_7\,(_8\,)_9\,)_{10}\,(_{11}\,(_{12}\,)_{13}\,)_{14}$$

We now consider each half separately by reconstructing the tree of partial results for each half and perform the same type of computations. The two trees to be reconstructed are in fact obtained by reconstructing the whole tree but disregarding its root. These two subtrees are therefore, each located in a set of consecutively indexed PE's. This insures that after the second iteration (at which time we have 4 subproblems) the cost of performing the required computations is reduced by a factor of 2. Our algorithm terminates in $\log n$ iterations when every left bracket at position i will have its matching right bracket in position $i+1$ and has complexity $O(\sum_{i=0}^{\log n-1} \sqrt{n}/2^i) = O(\sqrt{n})$.

The final result for our example is:

$$(_{1}\,)_{16}\,(_{2}\,)_{15}\,(_{3}\,)_{6}\,(_{4}\,)_{5}\,(_{7}\,)_{10}\,(_{8}\,)_{9}\,(_{11}\,)_{14}\,(_{12}\,)_{13}$$

Chapter 6

Finding Euler tours on feasible SIMD computers

6.1 Introduction

In previous chapters we dealt with explicitly tree-structured problems and showed that known techniques used to solve them on the P-RAM model can be efficiently implemented on some feasible machines. Here, we deal with the implementation of a generally useful tool namely Eulerian tour finding in a graph, which in contrast is not explicitly tree-structured.

Finding an Eulerian tour (circuit) in a graph is one of the oldest problems in graph theory [Gi85]. The problem is to find a way of traversing every edge exactly once in a tour of the graph. Besides its own right, the importance of the solution to this problem is further stressed in P-RAM algorithms such as those for finding a maximal matching in a graph [IS86] or computing the ear decomposition of a 2-edge connected graph [KR91].

In a sequential fashion the existence of an Eulerian tour in a graph (or the ***Eulerian property*** of the graph) is easy to decide and there are several algorithmic ideas to solve the problem. However, these sequential algorithms such as the linear time algorithms of [EJ73] and [B62] are not easy to parallelise.

Solutions in parallel environments appeared in [AV84] and [AIS84] but on models such as the $CRCW$ P-RAM. No known attempt has been made on more realistic machines such as for instance the MCC^2. In section 6.4, we show that such a problem can also be solved in efficient parallel time on such a machine by simulating the procedures used by Awerbush *et al* [AIS84].

In the following two sections, we state some used definitions and briefly review the P-RAM solutions of [AV84] and [AIS84] for the Eulerian tour problem.

6.2 Eulerian property of graphs

An ***Eulerian graph*** is an undirected graph or digraph, which contains an Eulerian circuit.

An undirected graph $G = (V, E)$ is Eulerian if and only if it is connected and all vertices are of even degree.

A digraph $H = (V, E)$ is an Eulerian digraph if and only if its underlying graph is connected and $\forall u \in V$ we have $d_{in}(u) = d_{out}(u)$, where $d_{in}(u)$ represents the in-degree of vertex u and $d_{out}(u)$ represents its out-degree.

A partition of the set of edges of a digraph $H = (V, E)$ to (edge-disjoint) circuits $(C_1, C_2, \ldots, C_k)$ is an Eulerian partition of G if each edge appears exactly once in its circuit. An Euler partition is unique if for every edge $e \in E$, a unique 'successor' is specified. A successor of e can be any edge emanating from a vertex which e enters. Any one-to-one mapping of entering edges to leaving edges in each vertex can determine an appropriate successor for each edge.

6.3 Parallel approaches to solve the Eulerian circuit problem

The parallel algorithms of [AV84] and [AIS84] use the same strategy. They both start by finding an Euler partition of the input graph, then find a way to stitch the edge disjoint cycles obtained. Their complexities (for a given graph $G = (V, E)$) are respectively $O(\log |E|)$ using $|E|$ PE's and $O(\log |V|$ using $|V| + |E|$ PE's on the $CRCW$ PRAM model of computation. Both algorithms take a directed Eulerian graph as input. To find an Eulerian circuit of an undirected Eulerian graph, both sets of authors use a preprocessing phase to orientate the graph. This preprocessing ensures that the oriented graph is Eulerian.

6.3.1 Outline of algorithm 1

The algorithm of Atallah and Vishkin [AV84] proceeds as follows: After partitioning the edges of the input graph into edge-disjoint circuits, this algorithm finds a spanning tree of a suitably defined auxiliary graph. Then an Eulerian circuit of the spanning tree of the auxiliary graph is found (such a problem is easier to solve using the Euler tour technique of Tarjan and Vishkin [TV85]). Finally, the Eulerian circuit of the spanning tree is expanded to an Eulerian tour of the input graph. A high level description of the algorithm is as follows :

1. Find an Euler partition of the input graph $G = (V, E)$ by means of lexicographical sorting.

2. Construct an auxiliary undirected bipartite graph $G_B = (W, E)$ defined as follows : G_B has two sets of vertices : circuit vertices and real vertices. The circuit vertices are the circuits obtained from step 1 and every vertex of G is a real vertex. There is an edge between a real vertex and a circuit vertex if and only if the (corresponding) vertex lies on the corresponding circuit in G.

3. Find a spanning tree $T = (W, F)$ of G_B and replace each edge T by two antiparallel edges to obtain an Euler digraph T'.

4. Find an Euler circuit of T' and use it to guide the stitching of the circuits found in step 1 into an Euler circuit.

6.3.2 Outline of algorithm 2

The interesting feature of the algorithm of Awerbush *et al* [/AIS84] is that it uses two other algorithms that seem not to be closely related to our problem namely ones that find the connected components and a spanning tree of an undirected graph. This algorithm starts by finding an Euler partition of the input graph, then using a connected components algorithm, a so called circuit-graph is computed. A spanning tree is then extracted from this circuit graph and the weights of its edges are used to modify the Euler partition so that the result is an Eulerian tour of the input graph. The steps performed by this algorithm can be briefly described as follows :

1. Generate an Euler partition P of the input graph $G = (V, E)$ by means of lexicographical sorting.

2. Name the circuits of P. i.e tell each edge to which circuit of P it belongs (By means of a connected component algorithm).

3. Construct a circuit graph C_G defined as follows: The vertices of C_G are the circuits obtained from step 1 and there exists an edge (link) between two vertices (circuits) if they have a common vertex (of G). Edges are labeled (e_1, e_2) where e_1 and e_2 are the edges entering that same vertex.

4. Find a (weighted) spanning tree T of C_G.

5. Execute the switch operations on T i.e. exchange the successors of every two edges labeling an edge of T.

6.4 Algorithm on MCC^2

The Algorithm presented is an adaptation of that of Awerbush *et al.* [AIS84]. The initial configuration for our problem is that each edge (i, j) of the directed Eulerian graph $G = (V, E)$ ($|E| = m$) will be stored in one PE of the $(\sqrt{m} \times \sqrt{m})$ MCC^2 (with shuffled row major indexing). Our implementation consists of the following six steps which are illustrated in detail in section 6.4.1.

Step 1. Find an Euler partition by means of lexicographical sorting.

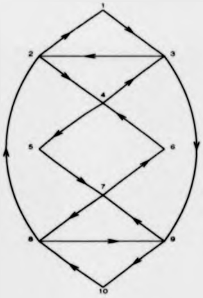
Step 2. Construct the line graph of the graph consisting of the edge-disjoint cycles obtained in step 1.

Step 3. Find the connected components of the line graph obtained from step 2.

Step 4. Construct a circuit graph $C_G = (P, L)$ where P is the set of edge-disjoint cycles obtained in step 1 and L is a set of links defined as follows: There is a link (C_i, C_j) where $C_i, C_j \in P$ if C_i and C_j have a common vertex.

Step 5. Select switches.(to stitch the edges disjoint cycles obtained in step 1) by finding a spanning tree of C_G.

Step 6. Execute the switching operations on T to finally obtain an Euler tour of the input graph.



(a)

PE(i)	EDGE(i)
1	(2,1)
2	(4,3)
3	(4,5)
4	(7,8)
5	(9,7)
6	(7,6)
7	(9,10)
8	(6,4)
9	(8,2)
10	(3,9)
11	(5,7)
12	(1,3)
13	(3,2)
14	(10,8)
15	(2,4)
16	(8,9)

(b)

Figure 6.1: A 16-edge Eulerian digraph

6.4.1 Detailed description:

We will be illustrating the computations of each step using the graph of figure 6.1(a) The input consists of a list of edges initially stored in register $EDGE(i)$ for each PE(i) (figure 6.1(b)). Furthermore, every step of our implementation will require the use of an additional number of registers (memory locations) bounded by the maximum vertex degree of our input graph.

Step 1. The aim in this step is to partition the input graph into edge disjoint cycles i.e. find an Euler partition. The following computations perform a 1-to-1 mapping of the entering edges to exiting edges for each vertex [GR88]:

1.1 Sort the edges in $EDGE(i)$ according to the following lexicographical order: $(j,k) < (l,m)$ if $k < m$ or ($k = m$ and $j < l$).

1.2. Copy vector $EDGE(i)$ into vector $SUCCESSOR(i)$.

1.3. Set pointer $P(i) = i$ for all i

1.4. Sort records ($SUCCESSOR(i)$, $P(i)$) on key $SUCCESSOR(i)$ according to the following lexicographical order : $(j,k) < (l,m)$ if ($j < l$) or ($j = l$ and $k < m$). Each edge of EDGE will recognize its successor by the pointer $P(i)$.

Step 1 finds an Euler partition of G (i.e. a set of edge-disjoint circuits) and can be achieved in $O(\sqrt{m})$ time on a ($\sqrt{m} \times \sqrt{m}$) MCC^2 using the sorting algorithms of [NS79] or [TK77]. Table 6.1 shows the contents of registers $EDGE(i)$, $SUCCESSOR(i)$ and $P(i)$ for all PE's after applying the above computations and figure 6.2 shows the corresponding Euler partition for our example graph. The output from this step will mainly constitute the input for a connected components algorithm to achieve circuit identification.

Step 2. The 'special' line graph LG to be constructed is defined as follows : For each edge of G there is a vertex of LG and two vertices of LG are adjacent if one of the corresponding is a successor of the other in the Euler partition. The idea behind this step is to prepare an input to the connected components algorithm of [NS81] that runs in $O(\sqrt{m})$ time for a graph where the maximum vertex degree is $d = 2$. The circuits obtained by the Euler partition can have a maximum vertex

PE(i)	EDGE	SUCCESSOR	P(i)
1	(2,1)	(1,3)	4
2	(3,2)	(2,1)	1
3	(8,2)	(2,4)	6
4	(1,3)	(3,2)	2
5	(4,3)	(3,9)	14
6	(2,4)	(4,3)	5
7	(6,4)	(4,5)	8
8	(4,5)	(5,7)	10
9	(7,6)	(6,4)	7
10	(5,7)	(7,6)	9
11	(9,7)	(7,8)	12
12	(7,8)	(8,2)	3
13	(10,8)	(8,9)	15
14	(3,9)	(9,7)	11
15	(8,9)	(9,10)	16
16	(9,10)	(10,8)	13

Table 6.1

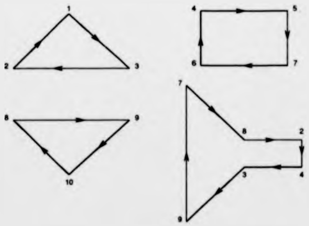


Figure 6.2: An Euler partition of the graph of figure 6.1

degree greater than 2 and therefore if it is proceeded otherwise the complexity of applying the algorithm of [NS81] to determine for each edge the circuit it belongs would be $\sqrt{n} \times \log n$ (as seen in chapter 4) whereas our goal is to stay as close as possible to $O(\sqrt{n})$ complexity overall. Moreover, the algorithm of [NS81] works on adjacency lists and therefore a refinement is required.

After step 1 each PE of the MCC^2 contains an element of $EDGE(i)$ and the pointer $P(i)$ specifying its successor in the Euler partition. Note that elements in $SUCCESSOR(i)$ need not to be kept. The correspondence edge - vertex will be done as follows : The edge (in $EDGE(i)$) contained in PE(i) will be named vertex v_i, and the adjacency list for that vertex is stored in registers $ADJ(i,0)$ and $ADJ(i,1)$.

The pointer $P(i)$ associated with the successor of the edge concerned will be $ADJ(i,0)$. $ADJ(i,1)$ will store the label of the vertex (edge) of which vertex (edge) i is the successor. This is done by sorting records $(v_i, P(i))$ on key $P(i)$ and the value to be kept in $ADJ(i,1)$ is the vertex label of the record just sorted. Columns 2 and 4 of table 6.2 show (for our example) the contents of $EDGE$ and P for every PE(i) and columns 3, 5 and 6 show respectively the labeling of the edges and contents of $ADJ(i,0)$ and $ADJ(i,1)$. Clearly, this step is achievable in $O(\sqrt{m})$ parallel time as it mainly involves sorting procedures.

Step 3. This step consists of applying the algorithm of [NS81] that finds the connected components of an undirected graph (max vertex degree ≤ 2) given by its adjacency list representation (in our case this is given by $ADJ(i,0)$ and

PE(i)	EDGE(i)	vertex(i)	P(i)	ADJ(i,0)	ADJ(i,1)
1	(2,1)	1	4	4	2
2	(3,2)	2	1	1	4
3	(8,2)	3	6	6	12
4	(1,3)	4	2	2	1
5	(4,3)	5	14	14	6
6	(2,4)	6	5	5	3
7	(6,4)	7	8	8	9
8	(4,5)	8	10	10	7
9	(7,6)	9	7	7	10
10	(5,7)	10	9	9	8
11	(9,7)	11	12	12	14
12	(7,8)	12	3	3	11
13	(10,8)	13	15	15	16
14	(3,9)	14	11	11	5
15	(8,9)	15	16	16	13
16	(9,10)	16	13	13	15

Table 6.2

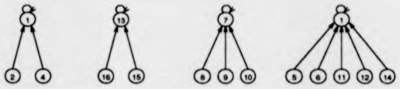


Figure 6.3: Output of the connected components algorithm

$ADJ(i,1)$ for every vertex i). After finishing all vertices belonging to the same component will point to that vertex of least index belonging to this component, that is, every component is named after its vertex of least index. This information will be stored in registers $C(i)$ (we will refer to this information as C_k where $k = C(i)$). For our example, figure 6.3 shows the output of the connected component algorithm (a set of reduced trees) and table 6.3 shows the sets of edges (renamed) and the identifier of the circuit they belong to. The complexity of this step is $O(\sqrt{m})$ [NS81].

PE(i)	EDGE(i)	Circuit identifier
1	(2,1)	1
2	(3,2)	1
3	(8,2)	3
4	(1,3)	1
5	(4,3)	3
6	(2,4)	3
7	(6,4)	7
8	(4,5)	7
9	(7,6)	7
10	(5,7)	7
11	(9,7)	3
12	(7,8)	3
13	(10,8)	13
14	(3,9)	3
15	(8,9)	13
16	(9,10)	13

Table 6.3

Step 4. Now that each edge knows to which circuit it belongs, we can construct the so-called circuit graph C_G. We require for every PE $2d$ additional (d = maximum in-degree of vertices in G) memory locations or registers $D(i,j)$ and $C(i,j)$, $(1 \leq i \leq n,\ 0 \leq j \leq d-1)$ (n is the number of vertices in our graph) to store the in-edges of each vertex of G along with their circuit identifiers. The strategy is to compress such records into the $2d$ locations of the PE with the same index that the vertex these edges ($EDGE$) enter (we assume that such a correspondence exists). For our example the edges $(3,2)$ and $(8,2)$ entering vertex 2 will be stored in $D(2,0)$ and $D(2,1)$ of PE(2) and $C(2,0)$ and $C(2,1)$ will respectively contain the values 1 and 3.

The above is achieved by sending the records consisting of (the edge (i,j) and the circuit identifier $C(i)$) stored in the PE's with an index $\neq j$ to the PE with

PE(I)	D(I,0)	C(I,0)	D(I,1)	C(I,1)
1	(2,1)	1	_	_
2	(3,2)	1	(8,2)	3
3	(1,3)	1	(4,3)	3
4	(2,4)	3	(6,4)	7
5	(4,5)	7	_	_
6	(7,6)	7	_	_
7	(5,7)	7	(9,7)	3
8	(7,8)	3	(10,8)	13
9	(3,9)	3	(8,9)	13
10	(9,10)	13	_	_
11	_	_	_	_
12	_	_	_	_
13	_	_	_	_
14	_	_	_	_
15	_	_	_	_
16	_	_	_	_

Table 6.4

index j. Such an operation is just an *RAW* (Random Access Write) where many PE's are trying to write into the same PE but onto different registers. This can be performed by the use of the routing algorithm of Nassimi and Sahni [NS80] either by compacting the many requests to write into the same PE. Table 6.4 shows the contents of the D registers for each PE. The circuit graph C_G constructed is identified as follows: Its vertices are the circuit identifiers C_k's stored in registers $C(i,j)$'s and there exists a link between two such vertices every time that two edges stored in the D locations of every PE are combined (equivalent to two edges entering the same vertex).

As for an example, consider the following two edges: $(2,4)$, $(6,4)$ (stored in $D(4,0)$ and $D(4,1)$ of PE(4)) and belonging respectively to circuits C_3, C_7. Then this implies the existence of the link (C_3, C_7) in C_G and labelled $((2,4),(6,4))$.

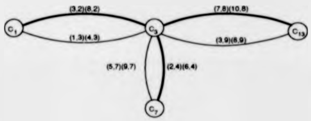


Figure 6.4

However, in the general case the number of links in the connected circuit graph C_G can be reduced by the use of an observation in [AIS84] stating that a connected subgraph of C_G will still lead to the same result. Therefore, the links of C_G that are considered are those with labels stored in locations $D(i,j)$, $D(i,j+1)$, $0 \leq j \leq d-2$ instead of all possible combinations. In figure 6.4 we illustrate the circuit graph CG for our example. The complexity of this step is dominated by the cost of invoking the routing algorithm of Nassimi and Sahni [NS80] to perform an RAW operation where at most d PE's attempt to write into the same location.

Step 5. This step consists of selecting a set of links of the "reduced" circuit graph, previously computed, such that after properly exchanging the successors of their labels in the Euler partition , an Euler tour of the input graph is obtained. From our previous example, if the link $((2,4),(6,4))$ is selected then this operation means the exchange of the successors of the edges $(2,4)$ and $(6,4)$ by simply modifying the P pointers.

One way of finding the set of links needed is to find any spanning tree of the subgraph of C_G [AIS84]. For this purpose, we make use of the efficient algorithm of Atallah and Kosaraju [AK84] for finding a minimum spanning tree of an undirected graph given by its adjacency matrix on a MCC^2.

The number of circuits in an Euler partition of an Eulerian graph with m edges is $\leq m/3$. Thus a $(\sqrt{m} \times \sqrt{m})$ MCC^2 could easily store the adjacency matrix of C_G. To construct the adjacency matrix of the graph C_G we proceed as follows : Every PE stores at most d records consisting of the edges entering the vertex with the same label and their circuit identifiers. The circuits have been named according to the output of the connected components algorithm and if we suppose the existence of say k circuits, their labels are in the range $[1 \,..\, m]$ (m number of edges in the input graph). But what we require is an identification of the circuits within the range $[1 \,..\, k]$ so that a simple routing operation will give us the adjacency matrix of our circuit graph. To change the range $[1 \,..\, m]$ to $[1 \,..\, k]$ we execute the following :

```
**for all** $i$, $1 \leq i \leq n$ **in parallel do**
**for all** $j$, $0 \leq j \leq d-1$ **in parallel do**
$M(C(i,j)) \leftarrow C(i,j)$ ($M$ is a new register required)
```

Many PE's will attempt to write into the same M location, but we do not care as they will be attempting to write the same data item. We then confine (compress) the M locations to an area of successively indexed PE's by using, for instance, a sorting procedure and can easily obtain a ranking of our circuits within the desired range i.e. $[1 \,..\, k]$.

Now that each circuit has been renamed, what is required is to update the information for all edges. We achieve this by distributing the new ranks (M) to the positions they were stored at before the compression and and then we let all the PE's read from those positions to finally update their information.

Now our task is to construct the weighted adjacency matrix W' of C_G. This is simply achieved by the following instruction :

```
for all $i$, $1 \leq i \leq m$ in parallel do
    for $j = 1$ to $d - 1$ do begin
        $W(C(i,j) \leftarrow C(i,j+1))$;
        $(D(i,j) \leftarrow D(i,j+1))$
    end
```

Having constructed W', we invoke the minimum spanning tree procedure with the convention that: $((a,b),(c,b)) \leq ((d,e),(f,e))$ if $b \leq e$

The time complexity of this step is dominated by that of constructing the adjacency matrix W of our circuit graph and is thus $O(d\sqrt{m})$ due to some sequential handling. Figure 6.4 shows the minimum spanning tree (edges in bold lines) computed for our example graph.

Step 6. The output of the minimum spanning tree procedure is a set of ($S \leq m/3 - 1$) marked edges stored as special pairs (i,j)'s carrying their weights with them. In our case, this set is a collection of edges of the form $(C_i, C_j)(i, j \in [1..k])$ with the weights or labels (W's). What remains to be done now is for every PE storing links (edges) of the circuit graph C_G to proceed to exchange the successors of the labels of those belonging to the minimum spanning tree. To achieve this, we allow each PE to store $d-1$ additional data items consisting of the weights (W's) of the edges of the minimum spanning tree ($SPAN$ registers). We begin by routing these W's which are of the form $((k,v),(l,v))$ to the appropriate PE(v) (they will stored in registers $SPAN$). Again and as in step 4 this computation can be achieved using the routing algorithm of Nassimi and Sahni [NS81] by compacting the information involved.

The last computation to be performed which is the execution of the switching operations is achieved by first identifying sequentially (for each PE) the edges that are involved in these operations by searching the registers D and $SPAN$ and then by modifying the P pointers accordingly, that is, exchanging the successors of the edges in $SPAN$. Table 6.5 illustrates the results of changing the adequate P pointers for our example. For each PE(i), these internal operations take respectively $O(d)$ and $O(d^2)$ sequential time to perform. The overall time complexity of achieving this step is $O(d\sqrt{m})$.

It is easy to see that the overall time complexity of the operations advocated for the implementation of the P-RAM algorithm of [AIS84] is $(O(d\sqrt{n})$. For architectures such as the PSC the same operations can be performed and will lead to $O(\log^3 n)$. Such a time complexity is dominated by that of finding the spanning tree and the connected components of a graph using the $O(\log^3 n)$ algorithms of [NM82].

PE(i)	EDGE(i)	SUCCESSOR(i)	P(i)
1	(2,1)	(1,3)	4
2	(3,2)	(2,4)	6
3	(8,2)	(2,1)	1
4	(1,3)	(3,2)	2
5	(4,3)	(3,9)	14
6	(2,4)	(4,5)	8
7	(6,4)	(4,3)	5
8	(4,5)	(5,7)	10
9	(7,6)	(6,4)	7
10	(5,7)	(7,6)	9
11	(9,7)	(7,8)	12
12	(7,8)	(8,9)	15
13	(10,8)	(8,2)	3
14	(3,9)	(9,7)	11
15	(8,9)	(9,10)	16
16	(9,10)	(10,8)	13

Table 6.5

Chapter 7

Conclusions

This thesis has investigated the implementation of many techniques and basic tools which evolved from research within the natural P-RAM model. As a result, many efficient algorithms designed within this model were shown to be implementable in optimum time on more feasible machines and particularly on the 2-dimensional mesh-connected model.

In chapters 2 and 3 we have surveyed a set of aids that frequently occur the ever growing literature on parallel computation. For instance, by showing that many of these aids can be implemented efficiently on an 2-dimensional mesh-connected computer we showed or indicated that many of the NC algorithms and utilities in Vishkin's structural algorithmics [Vi91](chapter 3) retain their inter-dependence and that their time complexities frequently translate to $O(\sqrt{n})$ which is optimal.

In chapter 4, some important recursively reducible problems were categorised and shown to be efficiently implementable on feasible machines. Problems such as polynomial evaluation, list ranking and expression evaluation were shown to be possible in $O(\sqrt{n})$ on the 2-dimensional mesh-connected computer for inputs of length n. Moreover, for many of the problems treated in chapter 4 which had a poor processor utilisation, we suggested a way for improving it by the use of pipelining.

Chapter 5, a natural extension of the previous chapter, showed how the balanced binary tree technique (also commonly employed in the design of efficient algorithms on the P-RAM model) can be effectively utilised (to solve problems of size n and implying a balanced binary tree with n leaves) on a 2-dimensional mesh-connected computer with $(\sqrt{n} \times \sqrt{n})$ PE's. Such a utilisation is likely to yield optimal implementations of many P-RAM algorithms (based on the balanced binary tree) on the 2-dimensional mesh-connected computer that run in $O(\sqrt{n})$ time using n PE's (rather than $\approx 4n$, as implied by the H-tree approach). As examples we showed in particular how, if the input is in the form of a string (of the symbols making up the expression) stored in an array, the non-trivial problems of evaluating arithmetic expressions, evaluating algebraic expressions with a carrier of constant bounded size and parsing expressions of both bracket and input driven languages have efficient solutions on the 2-dimensional mesh-connected computer.

Dealing with non-tree structured problems it was shown in chapter 6 that using an initial configuration of one edge per PE for a digraph $G = (V, E)$ with m edges and where the vertices have maximum in-degree d, the Eulerian circuit problem can be solved on a $(\sqrt{m} \times \sqrt{m})$ MCC^2 in $O(d\sqrt{m})$ parallel time. The same operations devised on the 2-dimensional mesh-connected computer lead to an $O(\log^3 n)$ solution on the perfect shuffle computer. Our solution was based on that of Awerbush *et al.* [AIS84] and is likely to be equivalent in complexity terms to a solution based on the method of Atallah and Vishkin [AV84]. One interesting fact though is whether combining techniques from both P-RAM algorithms would lead to another solution for the Eulerian circuit problem. It is likely that applying the Euler tour technique (used by [AV84]) to the spanning tree of the circuit graph in [AIS84] will yield the same solution.

The studies of this thesis concerned general techniques for P-RAM algorithm implementation on distributed memory machines. These investigations showed that many of these techniques can be usefully and optimally automated in the guise of methods and programs on such machines. This is particularly true for the 2-dimensional mesh-connected computer where, at present, completely general P-RAM emulation is not well understood.

Bibliography

[A83] Atallah, M. J., *Finding Euler tours in parallel*, Proceedings of the 7th Annual Conference on Information Sciences and Systems, 1983, pp. 685-689.

[A85] Akl, S. G., *Parallel Sorting Algorithms*, Academic Press, Orlando, Florida, 1985.

[A89] Akl, S. G., *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[A82] Aleliunas, R., *Randomised parallel communication*, $ACMSIGACT-SIGOPS$ Symposium on principles of distributed computing, August 1992, pp. 60-72.

[AH85] Atallah, M. J. and Hambrusch, S. E., *Solving tree problems on a mesh-connected processor array*, Proceedings of 26th annual IEEE symposium on Foundations of Computer Science, 1985, pp. 222-231.

[AHU74] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachussets, 1974.

[AIS84] Awerbush, B., Israeli, A. and Shiloach, Y., Finding Euler circuits in logarithmic parallel time, Proceedings of 16th ACM symposium on Theory of Computing, May 1984, pp.249-247.

[AK84] Atallah, M. J. and Kosaraju, R., *Graph problems on a mesh-connected processor array*, Journal of the ACM, Vol. 31, No. 3, 1984, pp. 649-667.

[AKS83] Ajtai, M., Komlos, J., and Szemeredi, E., *An $O(\log n)$ sorting network*, Proceedings ACM Symposium on Theory of Computation, April 1983, pp. 1-9.

[AL81] Agerwala, T. and Lint, B., *Communication issues in the design and analysis of parallel algorithms*, IEEE Transactions on Software Engineering, Vol. SE-7, No. 2, March 1981, pp.174-188.

[AV84] Atallah, M. J. and Vishkin, U., *Finding Euler tours in parallel*, Journal of Computer Systems Science, 29, 1984, pp. 330-337.

[B62] Berge, C., *The theory of graphs and its applications*, Wiley, New York, 1962.

[BK80] Brent, R. P., and Kung, H. T., *On the area of binary tree layouts*, Information Processing Letters, 11, 1980, pp. 46-48.

[BH82] Borodin, A. and Hopcroft, J., *Routing, merging and sorting on parallel models of computation*, Proceedings of 14th Annual ACM Symposium on Theory of Computing, , 1982, pp. 338-344.

[BV85] Bar-On, I. and Vishkin, U., *Optimal generation of a tree form*, ACM Transactions on Programming Languages and Systems, Vol. 7, 1985, pp. 348-357.

[BS90] Barnard, D. T. and Skillicorn, D. B., *Pipelining tree-structured algorithms on SIMD architectures* Information Processing Letters, 35, 1990, pp. 79-84.

[C86] Cole, R., *Parallel merge sort*, Proceedings of the 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 511-516.

[CKS81] Chandra, A. K., Kozen, D. C. and Stockmeyer, L. J., *Alternation*, Journal of the ACM, Vol. 28, 1981, pp. 114-133.

[CP90] Cypher, R. and Plaxton, C. J., *Deterministic sorting in nearly logarithmic time on the hypercube and related computers*, Proceedings of the 22nd ACM Symposium on Theory of Computing, 1990, pp. 193-203.

[E88] Ebert, J., *Computing Eulerian trails* Information Processing Letters, 28, 1988, pp. 93-97.

[EJ73] Edmonds, J., and Johnson, E. L., *Matching, Euler tours and the Chinese postman*, Mathematical Programming, 5, 1973, pp. 88-124.

[F66] Flynn, M. J., *Very high speed computing systems*, Proceedings IEEE 54, 1966, pp. 1901-1909.

[FW78] Fortune, S. and Willie, J., *Parallelism in Random Access Machines*, Proceedings of the 11th Annual ACM Symposium on Theory of Computing, 1978, pp. 114-118.

[G82] Goldshlager, L. M., *A universal interconnection pattern for parallel computers*, Journal of the ACM, Vol. 29, 1982, pp. 1073-1086.

[Gi85] Gibbons, A. M., *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, 1985.

[Gi91] Gibbons, A. M., *A tutorial introduction to distributed memory models of parallel computation*, Research Report 185, Department of Computer Science, University of Warwick, 1991.

[GKT79] Guibas, L. J., Kung, H. T. and Thompson C. D., *Direct VLSI implementation of combinatorial algorithms*, Proceedings of the Conference on VLSI, Caltech, Pasadena, California, January 1979, pp. 509-525.

[GR88] Gibbons, A. M. and Rytter, W., *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.

[GR89] Gibbons, A. M. and Rytter, W., *Optimal parallel algorithms for dynamic expression evaluation and context free recognition*, Information and Computation, Vol. 81, No. 1, April 1989, pp. 32-45.

[GRa92] Gibbons, A. M. and Ravindran, S., *Dense edge-disjoint embedding of complete binary trees in the hypercube*, Internal Report No. 223, Department of Computer Science, University of Warwick, 1992.

[KH86] Krishnan, M. S. and Hayes, J. P., *An array layout methodology for VLSI circuits*, IEEE Transactions on Computers, Vol C-35, No. 12, December 1986, pp. 1055-1067.

[KL85] Kindervater, G. A. P. and Lenstra, J. K., *An introduction to parallelism in combinatorial optimisation*, Research Report $OS - R8501$, Department of Operations Research and System Theory, Centre for Mathematics and Computer Science, Amsterdam, February, 1985.

[KR91] Karp, R. M. and Ramachandran, V., *Parallel Algorithms for Shared-Memory Machines*, In Handbook of Theoretical Computer Science, Volume A : Algorithms and Complexity, J van Leeuwen (ed.), 1991.

[L92] Leighton, F. T., *Introduction to parallel algorithms and architectures: Arrays-Trees-Hypercubes*, Morgan Kaufman, 1992.

[MR85] Miller, G. L. and Reif, J., *Parallel tree contraction and its applications*, Proceedings 26th Annual IEEE Symposium on Foundations of Computer Science, 1985, pp. 478-489.

[MS89] Miller, R. and Stout, Q. F., *Mesh computer algorithms for computational geometry*, IEEE Transactions on Computers, Vol. C-38, No. 3, March 1989, pp. 321-340.

[MP88] Maggs, B. M. and Plotkin, S. A., *Minimum-cost spanning tree as a path finding problem*, Information Processing Letters, 26, 1988, pp. 291-293.

[NM82] Nath, D. and Maheshwari, S. N., *Parallel algorithms for the connected* *components and minimal spanning tree problems*, Information Processing Letters 14, 1982, pp. 7-11.

[NS79] Nassimi, D. and Sahni, S., *Bitonic sort on a mesh-connected parallel computer*, IEEE Transactions on Computers, Vol. C-28, No.1, January 1979, pp. 2-7.

[NS80] Nassimi, D. and Sahni, S., *Finding connected components and connected ones on a mesh-connected parallel computer*, SIAM Journal on Computing, Vol. 9, No. 4, November 1980, pp. 745-757.

[NS81] Nassimi, D. and Sahni, S., *Data broadcasting in SIMD computers*, IEEE Transactions on Computers, Vol. C-30, No. 2, February 1981, pp. 282-288.

[O74] Orcutt, S. E., *Computer organization and algorithms for very high speed computations*, Ph. D. Thesis, Stanford University, 1974.

[P78] Preparata, F., *New parallel sorting schemes*, IEEE Transactions on Computers, Vol. C-27, 1978, pp. 669-673.

[Q87] Quinn, M. J., *Designing efficient algorithms for parallel computers*, McGraw-Hill, Singapore, 1987.

[QD84] Quinn, M. J., and Deo, N., *Parallel graph algorithms*, ACM Computing Surveys 16, 1984, pp. 319-348.

[RS90] Ranka, S. and Sahni, S., *Hypercube algorithms with applications to image processing and pattern recognition*, Springer Verlag, 1990.

[S80] Schwartz, J. T., *Ultracomputers*, ACM Transactions on Programming Languages and Systems, Vol. 2, 1980, pp. 484-521.

[S71] Stone, H. S., *Parallel processing with the perfect shuffle*, IEEE Transactions on Computers, Vol. C-20, February 1971, pp. 153-161.

[SSc88] Saad, Y. and Schultz, M. H., *Topological properties of hypercubes*, IEEE Transactions on Computers, Vol. C-37, July 1988, pp. 867-872.

[SS86] Schnorr, C. P. and Shamir, A., *An optimal sorting algorithm for mesh connected computers*, Proceedings of the 18^{th} ACM Symposium on Theory of Computing, 1986, pp. 255-263.

[SV81] Shiloach, Y. and Vishkin, U., *Finding the maximum, merging and sorting in a parallel model of computation*, Journal of Algorithms, Vol. 2., pp. 88-102, 1981.

[TK77] Thompson, C. D. and Kung, H. T., *Sorting on a mesh-connected parallel computer*, Communications of the ACM, Vol. 20, No. 4, April 1977, pp. 263-271.

[TV85] Tarjan, R. E., and Vishkin, U., *Finding biconnected components and computing tree functions in logarithmic parallel time*, SIAM Journal of computing, Vol. 14., 1984, pp. 580-599.

[TW91] Trew, A. and Wilson, G. (Eds) *Past, Present, Parallel - A survey of available parallel computers*, Springer-Verlag, 1991.

[U84] Ullman, J. D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, Maryland, 1984.

[V80] Valiant, L. G., *Experiments with a parallel communication scheme*, Proceedings 18th Allerton Conference on Communication, Control and Computing, 1980, pp. 802-811.

[V83] Valiant, L. G., *Optimality of a two-phase strategy for routing in interconnection networks*, IEEE Transactions on Computers, Vol. C-32, No. 9, September 1983, pp. 861-863.

[Vi91] Vishkin, U., *Structural parallel algorithmics*, Proceedings of the 18th ICALP, Springer-Verlag, pp. 363-380, 1991.

[VB81] Valiant, L. G. and Brebner, G. J., *Universal schemes for parallel communication*, Proceedings of the 13th ACM Symposium on Theory of Computing, 1981, pp. 263-277.

[W85] Wu, A., *Embedding of tree networks into hypercubes*, Journal of Parallel and Distributed Computing, 2, 3, 1985, pp. 238-249.

[WC90] Wang, B. and Chen, G., *Two-dimensional processor array with reconfigurable bus system is at least as powerful as CRCW model*, Information Processing Letters 36, 1990, pp. 31-36.

D

175498