Aspects of Practical Implementations of PRAM Algorithms

Somasundaram Ravindran

A thesis submitted for the
Degree of Doctor of Philosophy

Department of Computer Science
University of Warwick

December 1993

To my daughter Meera, with love.

Acknowledgments

I am very grateful to my supervisor, Dr Alan Gibbons, who introduced me to the area of parallel computation and provided encouragement and support in my research. His quick mind and knowledge of the field greatly helped me in my thinking, writing proofs, and making me realise what I was doing in a much broader context. He was also a constant source of support in non-technical matters.

I would also like to thank the many members of the Computer Science Department at the University of Warwick who provided a diversity of assistance over the past three years.

Above all I would like to gratefully thank my wife, Mythili. Without her constant help and support, this thesis would not have been possible.

Declaration

This thesis is submitted to the University of Warwick in support of my application for admission to the degree of Doctor of Philosophy. 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. Parts of the thesis appeared in the following papers in which my own work was that of a full pro-rata contributor:

1. S. Ravindran, B. Dessau and A.M. Gibbons, "An overview of PRAM to practical PRAM algorithmics", *Report 6.2.1, Parallel Universal Message-passing Architecture*, ESPRIT Project 2701 of the EC, 1991.
2. S. Ravindran and A.M. Gibbons, "Dense edge-disjoint embedding of binary trees in the hypercube", *Information Processing Letters*, Vol. 45, 321-325, 1993.
3. S. Ravindran and A.M. Gibbons, "Densely embedding the complete binary tree in communication networks", 9^{th} *British Colloquium for Theoretical Computer Science*, University of York, England, March 1993.
4. S. Ravindran, A.M. Gibbons and M.S. Paterson, "Dense edge-disjoint embedding of complete binary trees in interconnection networks", submitted to *Theoretical Computer Science*, December 1993.
5. S. Ravindran, N.W. Holloway and A.M. Gibbons, "Approximating minimum weight perfect matchings for complete graphs satisfying the triangle inequality", in *Proceedings of* 19^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, Springer-Verlag, 1993.

[illegible]

S. Ravindran 20th December 1993

Abstract

The PRAM is a shared memory model of parallel computation which abstracts away from inessential engineering details. It provides a very simple architecture independent model and provides a good programming environment. Theoreticians of the computer science community have proved that it is possible to emulate the theoretical PRAM model using current technology. Solutions have been found for effectively interconnecting processing elements, for routing data on these networks and for distributing the data among memory modules without hotspots. This thesis reviews this emulation and the possibilities it provides for large scale general purpose parallel computation. The emulation employs a bridging model which acts as an interface between the actual hardware and the PRAM model. We review the evidence that such a scheme can achieve scalable parallel performance and portable parallel software and that PRAM algorithms can be optimally implemented on such practical models. In the course of this review we presented the following new results:

1. Concerning parallel approximation algorithms, we describe an NC algorithm for finding an approximation to a minimum weight perfect matching in a complete weighted graph. The algorithm is conceptually very simple and it is also the first NC-approximation algorithm for the task with a sub-linear performance ratio.

2. Concerning graph embedding, we describe dense edge-disjoint embeddings of the complete binary tree with n leaves in the following n-node communication networks: the hypercube, the de Bruijn and shuffle-exchange networks and the 2-dimensional mesh. In the embeddings the maximum distance from a

leaf to the root of the tree is asymptotically optimally short. The embeddings facilitate efficient implementation of many PRAM algorithms on networks employing these graphs as interconnection networks.

3. Concerning bulk synchronous algorithmics, we describe scalable *transportable* algorithms for the following three commonly required types of computation; balanced tree computations, Fast Fourier Transforms and matrix multiplications.

Contents

Contents

Chapter 1

Introduction

A few years ago parallel computers could be found only in research laboratories. Due to the rapidly decreasing cost of processors, memory and communication, they are now available commercially. It is possible that, within a decade, parallel computation will dominate in all areas of computer science and its applications. A deep understanding of parallel computations is therefore highly desirable.

The complexity theory research community has developed a rich literature in the design and analysis of efficient parallel algorithms. To date, most of this work has been based on the *Parallel Random Access Machine* (PRAM). The PRAM model consists a number of processing elements and a (shared) memory. The processing elements operate in lock step synchrony and access a location in the shared memory in unit time (chapter 2 gives a detailed description of the PRAM model). The realistic issues are ignored by the PRAM model. The PRAM algorithms do not take account of the low level details of parallel computation, such as interprocessor or processors-to-memory communication, memory management and hardware failure.

For this reason, the PRAM provides a very simple and natural model for architecture independent parallel algorithm design.

One benefit of PRAM studies is an extensive list of *fast* parallel algorithms. By a *fast* algorithm we mean one that that takes polylogarithmic parallel time using a polynomial number of processors. Problems which can be solved by a *fast* algorithm are said to belong to the class NC. This class name is an acronym for Nick (Pippenger's) Class. The search for parallel solutions to place the problems in NC has demonstrated that entirely new algorithmic techniques are appropriate and it is usually a bad starting point to attempt to parallelise the best sequential algorithms. Design techniques and tools have therefore been developed for parallel computation which are completely different from the sequential domain. In fact, some of the commonly used methods in sequential algorithm design do not adapt well to parallelism. Moreover, many problems in the class P, i.e. the problems have polynomial time sequential solution, have been proven *P-complete*. If a problem is *P-complete* then it is very unlikely that the problem belongs to the class NC.

The primary interest of the study of parallel complexity is placing the problems in P into the class NC. In fact, there are problems in P but we do not know whether they lie in NC, and they have not been proven to be *P-complete*. However, for such problems it may be possible to find an NC approximation algorithm to the problems. For example, if the problem is a minimisation problem then we find a solution which is not minimum, but the solution is never greater than some factor of the optimal solution.

Although the PRAM model has been prominent in the development of the design and analysis of parallel algorithms, the unrealistic characteristics of the model may make us think that the model is not suitable for general purpose parallel computation. However, we shall see in this thesis that the PRAM model can be emulated by a feasible parallel computer. A feasible parallel computer is a number of processing elements interconnected by a network, where each processing element performs computational and/or memory operations and the processing elements may or may not have to be synchronised. The memory of a feasible parallel computer is distributed among memory modules.

In contrast to a PRAM, a (non local) memory access in a feasible parallel computer takes much longer than a local computation. This is because the data have to be transferred through an interconnection network. Further delays can occur due to congestion in the interconnection network and contention at the memory modules. Moreover, a PRAM assumes that the processors and the communications links are fault-free. Theoretical solutions have been found for routing data to the right processing element within a reasonable time, for distributing the data among memory modules so that the distribution will not slow down the computation, and for coping with processors and network failures.

At present, almost all of the parallel software designed for a realistic physical parallel model is not portable. This is because the algorithms are often developed for a particular network topology of fixed size. This software is not based on general principles and to date has not been written for a common virtual machine. Progress in technology suggests to us that rapid changes in parallel machines are still to come. Hence, current software will not have a long life time. The major challenge in parallel computing is developing architecture independent parallel software with an expected lifetime of several decades. We need an interface (or bridging) model between the software and hardware. Such a model will offer architecture independent software and will be compatible with technological evolution.

Using the theoretical solutions for routing and for memory management we can build a bridging model with virtual shared memory. A user can view this model as an extended PRAM, which hides hardware details from the user. The model is called *practical PRAM*. Recently a number of *practical PRAMs* have been proposed which variously take account of communication delay, contention and congestion, asynchrony and component failures. In this thesis we review these models and show that it is possible to develop an architecture independent software with a long life time.

The remainder of this thesis is organised as follows.

- Chapter 2 reviews the design techniques and tools of parallel computation, and provides the evidence for the significance of the PRAM model.
- Chapter 3 gives a novel approximation algorithm for a problem whose parallel complexity remains unknown. It is not known whether the problem lies in NC, and it has not been proven to be *P-complete*.
- Chapter 4 focuses on the realistic issues which are ignored by the PRAM model. If an interconnection network is to be used for general problem solving then it ought have certain desirable graph theoretic properties. Chapter 4 describes interconnection networks which satisfy these properties. Further-

more, chapter 4 reviews the theoretical solutions for routing, for memory management and for fault tolerance.

- The most commonly occurring structure in parallel computation is the complete binary tree. It is precisely because such logarithmic depth structures are used (either explicitly or implicitly) that polylogarithmic time complexities are attained for many PRAM algorithms. Chapter 5 shows that the complete binary tree can be embedded in the interconnection networks, the mesh, hypercube, de Bruijn and shuffle-exchange networks, so that an algorithmically important parameter, the maximum distance from a leaf to the root, is optimised asymptotically. Thus, $O(\log n)$ time PRAM algorithms which use the complete binary tree as the algorithmic structure can be translated to optimal time on the interconnection network.

- With the use of theoretical solutions which are described in chapter 4, chapter 6 demonstrates that there is no hindrance in designing practical parallel models of parallel computation. Moreover, this chapter shows that a PRAM algorithm can be implemented on the practical parallel model without any significant delay in run time.

- Chapter 7 shows that scalable transportable algorithms for practical parallel models can be written for certain basic tasks, balanced tree computations, Fast Fourier transforms and matrix multiplication.

Chapter 2

Classical PRAM Design

2.1 Introduction

The PRAM model of parallel computation is described in detail in section 2.2. At first glance, the PRAM model of computation might not appear to be suitable as a general model for designing and analysing parallel algorithms. For sequential computation, it has been of considerable advantage to deal with an abstraction of the von Neumann machine, namely the Random-Access Machine or RAM (see [3] for details of the model). Similar advantages justify the PRAM model:

- *Ease of use: Algorithms can be specified with little intricacy.*
- *Portability: PRAM algorithms do not need to take into account memory organisation, network topology or other hardware design attributes of real parallel machines, so that they eliminate obstacles to portability.*

- *Scalability: Typically the number of processors which are used can be increased in a natural way such that the speed of the computation retains the same functional dependence on the problem size.*

The main preoccupation of PRAM algorithm designers has been to place the problem in hand in the class NC. Various idealised models of parallel computation other than the PRAM model have been used in the study of parallel algorithms and their complexity. These include Boolean circuits and alternating Turing machines. The complexity class NC remains unchanged when defined by these models [138, 149]. This motivates the definition of the complexity class NC. Note that when using a family of Boolean circuits as a model of parallel computation, the family is usually required to satisfy a *logspace uniformity condition* [139]. A family of Boolean circuits is logspace uniform if there is a deterministic Turing machine can construct some standard encoding of the n^{th} circuit using $O(\log n)$ work space.

It is reasonable to seek an NC algorithm for a problem, if the problem can be solved sequentially in polynomial time. Let P be the class of decision problems that can be solved by a deterministic Turing machine within a polynomial number of sequential steps [3]. We can see that $NC \subseteq P$ by converting NC algorithms into sequential algorithms in the obvious way. A fundamental open question is whether every problem in P lies in NC. The *parallel computation thesis* states that "time bounded parallel machines are polynomially related to space bounded sequential machines" [18, 22, 43, 57, 122]. That is NC computations can be simulated by Turing machines using only polylogarithmic space. Thus, $P = NC$ would imply that P is contained in a class of problems that can be solved in polylogarithmic space by a sequential machine, which is considered very unlikely. This is why we believe that there exists problems which do not adapt well to parallelism. In fact, many problems have been proven to be in *P-complete*. For a list of *P-complete* problems see [59, 107]. If a problem is *P-complete* then the problem is unlikely to lie in NC. More formally, a problem $L \in P$ is said to be *P-complete* if every other problem in P can be transformed into L by a deterministic Turing machine using only log space (such a transformation is said to be a *log space reduction*). It follows that, if $L \in$ *P-complete* and $L \in NC$ then $P = NC$. It turns out that some of the commonly used methods in sequential algorithms are likely inherently sequential methods. Thus the design of parallel algorithms requires new paradigms and techniques.

In this chapter we show that, the benefit from the PRAM model is not only in the extensive list of efficient and parallel algorithms that have been designed, but also fundamental paradigms and design techniques and tools have emerged. These are usually completely different from the best known sequential solutions to the same problems and indicate new paradigms for parallel algorithm design. Sections 2.3 and 2.4 describe the techniques and tools respectively. In section 2.5 we review some graph algorithms, and show that these are useful not only in their own right for the problems they solve, but also as common subroutines in many parallel algorithms. Note that in this chapter we consider only graph algorithms. Of course, the PRAM can be and has been used to solve problems in many other areas. For example, survey papers [38] and [65] list references for *computational geometry* and *pattern matching* respectively. Many other references can be found in [42, 52, 68, 76, 103, 166].

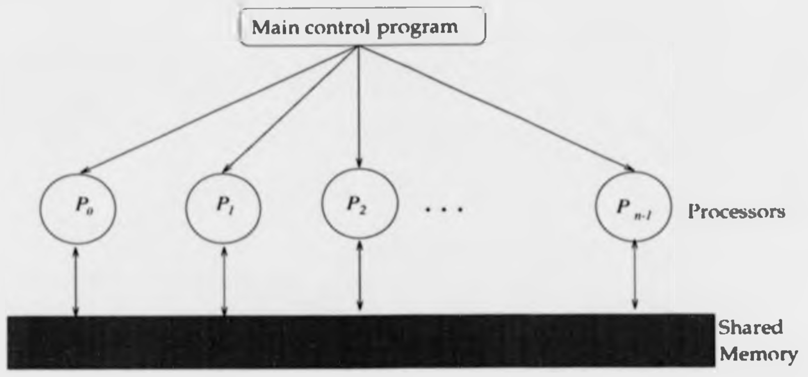


Figure 2.1:

2.2 The PRAM model

The PRAM model is an abstract shared memory model. It was introduced in [43, 169]. There are a number of processors working together and communicating through the shared memory. The processors synchronously execute the same program through the central main control (see figure 2.1). Although performing the same instructions, the processors can be operating on different data. Hence, such a model is a Single-Instruction, Multiple-Data model, namely an SIMD model. Each processor is a uniform-cost random-access machine or RAM with usual operations and instructions. The cost of arithmetical operations (addition, subtractions, equality predicate and so on) is constant. In one step each processor can access (either reading from it or writing to it) one memory location or execute a single RAM operation.

Memory access leads to variants of the model which allow or do not allow more than one processor to read or to write to the same memory location. For example the

EREW PRAM (exclusive read exclusive write parallel random access machine which allows no concurrent reads and no concurrent writes), the CREW PRAM (concurrent reads allowed but only exclusive writes) and the CRCW PRAM (in which both concurrent writes and concurrent reads are allowed). In general concurrent reads cause no logical errors. However, with concurrent writes additional rules are required to resolve the outcome. This leads to variants of the CRCW PRAM, namely the so-called *common*, *arbitrary* and *priority* variants. These resolves the write conflicts as follows: in the *common* variant all processors writing into the same location write the same value, in the *arbitrary* variant any one processor participating in a common write may succeed and the algorithm should work regardless of which one does, and in the *priority* variant there is a linear order on the processors and the minimum numbered processor succeeds in writing.

Any algorithm that works on an EREW PRAM works on a CREW PRAM, any algorithm that works on a CREW PRAM works on a *common* CRCW PRAM, and so on. Moreover, the list of variants of the PRAM model: EREW, CREW, *common* CRCW, *arbitrary* CRCW and *priority* CRCW, represents the PRAM models in increasing order of their power. But, they do not differ much in their power. The following theorem [164] indicates this. This justifies the class NC; the class remains unchanged regardless of variants of the PRAM model.

Theorem 2.2.1 *Any algorithm for a priority CRCW PRAM of p processors can be simulated by a EREW PRAM with the same number of processors and with the parallel time increased by a factor of $O(\log p)$.*

2.2.1 Concepts of efficient and optimal algorithms

In the PRAM model, the relevant complexity measure of an algorithm are the time for parallel computation and the number of processors used. The time-processor product of a PRAM algorithm is called the *work* of the algorithm.

Suppose a PRAM algorithm runs in time $t(n)$ by employing $p(n)$ processors for a problem size n, then the PRAM algorithm can be converted into a sequential algorithm of time equal to the work of the PRAM algorithm, $t(n) \times p(n)$. We can do this by simulating each parallel step of the PRAM algorithm on a sequential processor in $p(n)$ time units. This justifies the definition for an *optimal* algorithm. A PRAM algorithm for the problem in NC is *optimal* if the work of the algorithm is asymptotically equal to the fastest sequential computation time for the problem.

An optimal parallel algorithm achieves a high degree of parallelism. Analogously, a PRAM algorithm for the problem in NC is *efficient* if the work of the algorithm is within a polylogarithmic factor of the fastest sequential computation time for the problem. Designing an optimal algorithm on a CRCW PRAM is easier than on a CREW or EREW PRAM. This is because more parallelism can be expressed on a CRCW PRAM than on a CREW or EREW PRAM. The class of problems which have *efficient* algorithms, remains unchanged regardless of the PRAM model used (for example, see theorem 2.2.1). Thus our notion of *efficiency* is more robust than the notion of *optimality*.

Concerning speed of computation, one might expect that it is possible to discover parallel algorithms that run in constant time. Research on lower bounds for parallel computation indicates that this goal is unachievable for almost any interesting problems. This is because we require a lower bound of $\Omega(\log n / \log\log n)$ time for the parity problem of n bits on a *priority* CRCW PRAM with a polynomial number of processors [14]. This is because the parity problem depends on all the inputs. But the problem of computing the *OR* or *AND* of n Boolean variables can be done in constant time on a CRCW PRAM. Because, the decision of this problem only depends on one input. However, this computation requires $\Omega(\log n)$ on a CREW PRAM with no restriction on the number of processors [34]. Since any interesting problems such as the basic PRAM subcomputations of prefix computation and list ranking (described in section 2.4) are typically at least as hard as computing the parity of n bits or *OR* of n variables, we see that constant time parallel computation is not admissible for any interesting problems.

2.3 Basic PRAM techniques

A number of general techniques and principles of common use in the design of parallel algorithms are described in this section.

2.3.1 Balanced tree

The balanced binary tree is a fundamental structure in parallel computation. A tree is a connected graph containing no circuits, in which one vertex is distinguished as a root. In a tree any vertex with of degree one, unless it is the root, is called a leaf. A node is said to be an internal node if the node is not a leaf. If (x, y) is an edge of a tree such that x lies on the path from the root to y, then x is said to be the parent of

y and y is the child of x. A tree is balanced if each internal node has same number of children. A balanced tree is called complete binary tree if each internal node has two children and the number of nodes is $2n - 1$, where n is the number of leaves. The minimum distance between a leaf and the root in a complete binary tree is $\log n$, where n is the number of leaves.

It is precisely because such logarithmic depth structures are used (either explicitly or implicitly) that polylogarithmic time complexities are attained for many PRAM-algorithms. In the PRAM model, the balanced binary tree is employed as follows. Data for a problem are placed at the leaves, and each internal node corresponds to the computation of a subproblem. The sub problems are solved in bottom-up order (or in one or more sweeps up and down the tree), with those at the same level in the tree being computed in parallel.

For example, consider the problem of adding n numbers. Let $n = 2^m$ and A be an array of length $2n$. The numbers whose sum is to be found can be placed at the leaves of a tree, i.e. we store the n numbers in the locations $A(n), A(n+1), \ldots, A(2n-1)$. At each level of the tree, numbers are added together in pairs by different processors in parallel and the result sent to the next level as follows [52].

```
for $k \leftarrow m - 1$ step -1 to 0 do
    for all $j$, $2^k \leq j \leq 2^{k+1} - 1$, in parallel do $A(j) \leftarrow A(2j) + A(2j+1)$
```

A processor assigned to an internal node reads the values of the left child and the right child from the corresponding locations, then writes the sum of the values in the location corresponding to the internal node. If the corresponding location of an internal node is $A(j)$ then the corresponding locations of its left child and right child are respectively $A(2j)$ and $A(2j+1)$. At the end of the computation $A(1)$, the location corresponding to the root, stores the result. The depth of the tree is bounded by $\lceil \log n \rceil$ and so the computation time is $O(\log n)$ using $n/2$ processors. This problem clearly belongs to NC. It will be shown later how to reduce the number of processors to achieve optimal work measure.

2.3.2 Doubling

This technique is normally applied to an array or to a list of elements. Each element knows the location of the next element in the data structure. The computation proceeds by a recursive application of the calculation in hand to all elements over a certain distance (in the data structure) from each individual element. This distance doubles in successive steps. Thus after k stages the computation has been performed (for each element) over all elements within a distance of 2^k.

For example, consider an array A of n elements which specifies a set of rooted directed trees, a forest F. A location of the array $A(i) = j$ if j is the parent of i in a tree of F, for $1 \leq i \leq n$, and if i is a root then $A(i) = i$. Suppose, for each i, $1 \leq i \leq n$, we want to determine the root of the tree containing the node i. Let $s(i)$ be the successor of node i, i.e initially $s(i) = A(i)$, for $1 \leq i \leq n$. The successor of each node, $s(i)$, is replaced by the successor's successor, $s(s(i))$, in successive steps. If a processor is assigned to each array location then $O(\log h)$ steps are sufficient for this computation. Here h is the maximum height of any tree in the forest. Sometimes this technique is referred to as *pointer jumping*.

The doubling technique is not only applicable to arrays and lists. For example, it can be used to compute A^n of a Boolean matrix A. The computation can be performed in at most $2\lfloor\log_2 n\rfloor$ matrix multiplications. First we compute $A^2, A^4, A^8 \ldots A^k$ by squaring the matrix successively, where k is the largest number such that $k \leq n$ and k is a power of 2. Now we can compute A^n by multiplying together appropriate matrices of the form A^{2^l}, $0 \leq l \leq \lfloor\log_2 n\rfloor$.

2.3.3 Divide-and-conquer

The divide-and-conquer technique involves partitioning a problem into subproblems, solving the subproblems, and then combining the solutions to the subproblems to form the solution for the original problem. The methodology is recursive; that is, the subproblems themselves may be solved by the divide-and-conquer technique. This method is widely applicable in sequential computation. In a parallel setting the method requires that the subproblems at the same level of recursion can be independently computed in parallel and (in order to reduce the depth and therefore the computation time) are of similar size.

There are several examples of the divide-and-conquer technique applied in a parallel setting. Although the field of computational geometry is rather neglected in this thesis, we complete this description of divide-and-conquer with an application from this area. Given a finite set S of points in the plane, computing their ***convex hull***, $CH(S)$, is an important basic problem in computational geometry that arises in a variety of contexts [38]. $CH(S)$ is the smallest convex polygon containing all the points of S. A polygon $CH(U)$ is convex if, given any two points p and q in

$CH(U)$, the line segment whose endpoints are p and q lies entirely in $CH(U)$. A tangent of a convex polygon $CH(U)$ is a line passing through a vertex of $CH(U)$ such that $CH(U)$ lies entirely on one side of the line. A tangent is called upper (lower) common tangent between two convex polygon, $CH(U)$ and $CH(V)$, if the tangent is the common tangent such that $CH(U)$ and $CH(V)$ are below (above) it. $CH(S)$ can be constructed as follows:

1. *Sort the set S in some fixed direction (eg. in the x or y-direction). Let U be the first $|S|/2$ points in this sorted order, and V the remainder.*

2. *Recursively construct $CH(U)$ and $CH(V)$ in parallel.*

3. *Compute the upper and lower common tangents of $CH(U)$ and $CH(V)$.*

4. *Combine $CH(U)$ and $CH(V)$ by using the upper and lower common tangents of $CH(U)$ and $CH(V)$, to form $CH(S)$.*

2.3.4 Reducing the number of processors

Consider a computation A that can be done in t parallel steps. Let α_i be the number of primitive operations at step i. To run A directly on a PRAM in t parallel steps, the number of processors required is the maximum of the α_i, say m. Suppose we have $p < m$ processors. The i^{th} step can be computed in time $\lceil \alpha_i/p \rceil$, by partitioning the α_i operations into p groups and assigning a processor to each of the groups. For each of the p groups in parallel, each processor will be computing (in sequential style) for at most $\lceil \alpha_i/p \rceil$ time. Hence the total parallel time is no more than $t + \lceil \sum_{i=1}^{t} \alpha_i/p \rceil$

For example, consider the algorithm described earlier for computing the sum of n numbers using the balanced binary tree method. This executes in O($\log n$) time. Notice that $n/2$ processors are required for the first step. Suppose that we have $p < n/2$ processors. We can simulate the first step with p processors as follows. First we divide the n numbers into p groups, such that i^{th} group has elements indexed from $(i\lceil n/p\rceil)$ to $(((i+1)(\lceil n/p\rceil)) - 1)$ for $0 \leq i < p$. The first $(p-1)$ of these groups contain $\lceil n/p\rceil$ elements and the remaining group contain $n - (p-1)\lceil n/p\rceil$ elements. We assign a processor to each of the groups. For all of the p groups in parallel, each group now finds the sum in sequential style within its group, and the computation takes $\lceil n/p\rceil$ time. We have reduced the original problem of size n to a problem of size p, and the problem can be solved as described before in $O(\log p)$ time using the p processors. Thus, overall, we can find the sum of the n numbers in $\lceil n/p\rceil + \log p$ time using $p \leq n/2$ processors. Notice that if we set $p = n/\log n$ then we obtain a computation time of $O(\log n)$ and we thus have an optimal algorithm. The work of the algorithm is $O(n)$ and is same as the best known sequential algorithm.

This is an example of applying ***Brent's scheduling principle*** [20], and is often used in the design of efficient or optimal parallel algorithms. It should be noted that this simulation assumes that processor allocation is not a problem. We will see (in section 2.4.2) that this is sometimes a nontrivial task.

2.4 PRAM algorithmic tools

In this section we describe the algorithmic tools, known as *prefix computation* and *list ranking*, that have been found to be of wide use in the construction of many parallel algorithms. One can appreciate this from figure 2.2 [166], which illustrates how the solutions of some PRAM algorithms depend on the solutions of others. Such a diagram is an example of so-called *structural algorithmics*. Let P_1 and P_2 be the problems in figure 2.2 such that P_1 is above P_2. If there is an edge between P_1 and P_2 then the NC algorithm that solves P_2 is used to obtain an NC algorithm to solve P_1. Thus, if there is a path between P_1 and P_2 then all the solutions to the problems on the path are used to obtain an NC algorithm to solve P_1. As an example, the solution to the prefix sum problem is a subroutine in the solution to the list ranking problem, the solution to list ranking is a key subroutine to the so-called the Euler tour technique and so on.

2.4.1 Prefix computation

Given an array $[x_0, x_1, \ldots, x_{n-1}]$ of n elements together with an associative operator $*$, a prefix computation gives $S_i = x_0 * x_1 * \cdots * x_i$, for $1 \leq i \leq n-1$. There is a simple algorithm for performing prefix discovered by Ladner and Fisher [86]. The algorithm is perhaps easily understood with reference to the complete binary tree of the computation. Let n be a power of 2, otherwise we add a minimum number of dummy elements to achieve this. At the outset we store the n numbers at the leaves of the tree so that x_i is in the corresponding location of the i^{th} leaf for $0 \leq i \leq n-1$. The leaves are numbered from 0 to $n-1$ from the left to the right. Levels are

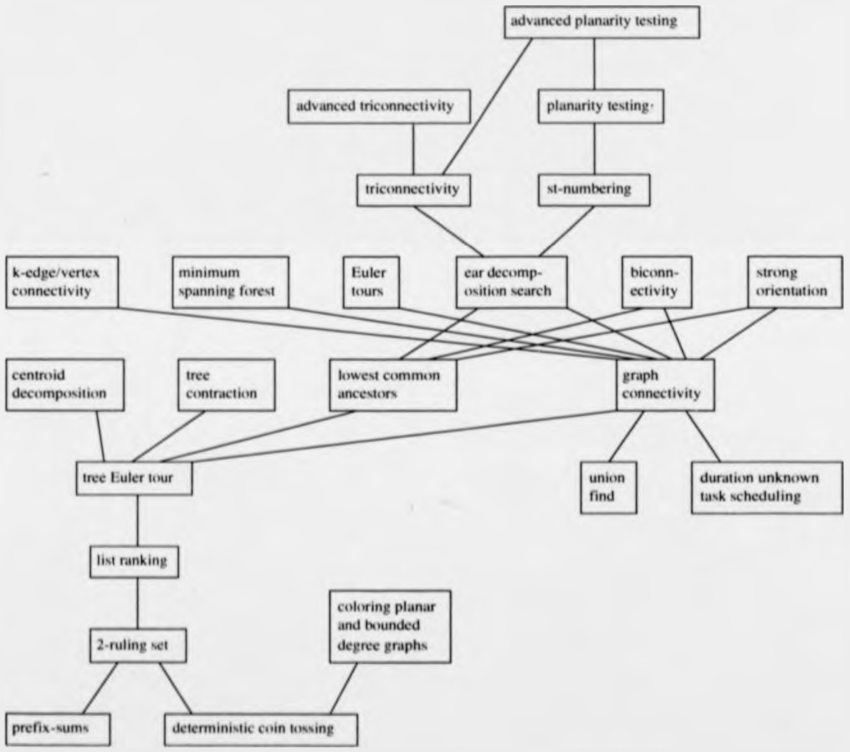

advanced planarity testing
advanced triconnectivity
planarity testing
triconnectivity
st-numbering
k-edge/vertex connectivity
minimum spanning forest
Euler tours
ear decomposition search
biconnectivity
strong orientation
centroid decomposition
tree contraction
lowest common ancestors
graph connectivity
tree Euler tour
union find
duration unknown task scheduling
list ranking
coloring planar and bounded degree graphs
2-ruling set
prefix-sums
deterministic coin tossing

Figure 2.2:

numbered from 0 to $\log n$, from the level of the leaves upwards. Let any node j of a complete binary tree cover the leaves from positions p to q (note if j is a leaf then $p = q$). Then let $A(j)$ and $B(j)$ be storage locations that will be used respectively to store the values of $x_p * x_{p+1} * \cdots * x_q$ and $x_0 * x_2 * \cdots * x_q$. The following code performs the desired computation.

```
for $level$=1 to $\log n$ do
    for all $j \in level$, in parallel do compute $A(j)$
$B(root) \leftarrow A(root)$
for $level = \log n - 1$ to 0 do
    for all $j \in level$ , in parallel do compute $B(j)$
```

At the end of computation $x_1 * \cdots * x_j$ is stored in the corresponding location $B(j)$ of the j^{th} leaf, for $0 \leq j \leq n-1$. For each node a processor can identify the level of the node and its children as in the computation of the sum of n numbers (see section 2.3.1). Moreover, the first phase of the computation from level 1 (one level above the leaves) to the root can be done as explained in section 2.3.1, by reading the values from left child and right child. In the second phase from the root to the leaves, $B(i)$ for the node i can be computed as follows. If the node i is the right child then $B(i)$ is $A(i\text{'s parent})$, otherwise $B(i)$ is $(B(i\text{'s parent}) * (A(i\text{'s sibling}))$. The computation can be run in $O(\log n)$ time on an EREW PRAM since there are no conflicts in memory accesses. At first it would seem that we need $O(n)$ processors to achieve this time. Since the input is stored in an array as already described (i.e in consecutive memory locations), we can easily achieve a optimal algorithm (i.e. the same time complexity with $O(n/\log n)$ processors) using Brent's scheduling principal as in section 2.3.4.

Given an array A of locations storing 0 or 1, the associated parity problem is to determine whether the number of 1s is even or odd in the array. This problem can be regarded as a special prefix computation problem. Performing such a computation on A will leave the result of the parity problem in the rightmost location of A. A lower bound of $\Omega(\log n / \log\log n)$ time is known for the parity problem on *a priority* CRCW PRAM with a polynomial number of processors. To match this lower bound for the prefix computation, Cole and Vishkin [28] described an optimal algorithm (different from the one described above) which runs in time $O(\log n / \log\log n)$ using $n \log\log n / \log n$ processors on a CRCW PRAM.

The fact that the prefix-sums problem appears at the bottom of figure 2.2 is meant to convey the basic role of this problem. It appears in many guises. For example, consider compacting a sparse array. Given an array of n elements, many of which are zero, we wish to generate a new array only containing the non-zero elements in their original order. This problem can be solved by assigning a value 1 to the non-zero elements, and performing the prefix sums using arithmetic addition. Such a computation calculates, for each non-zero element of the array, the position that such a non-zero element would have in the new array.

2.4.2 List ranking

Given a linked list, the list ranking problem is to calculate for each member of the list its relative position from the end of the list, i.e. its rank in the list. The importance of this problem was first identified by Wyllie [169]. An obvious solution to list ranking can be regarded as a prefix computation using addition of 1s within a pointer structure. A linked list is an alternative to an array in storing sequences of element in shared memory. In an array each element knows its address within the array whereas in a linked list an element does not know *a priori* its rank in the list.

Using the *pointer jumping* technique (as explained in section 2.3.2) we can solve the list ranking problem. Initially we set *distance(k) = 1* for each element k except the last for which we set *distance(k) = 0*. The last element can be easily determined by looking at the pointer's address, $s(i)$, because the last element uniquely points to itself. The algorithm is then described as follows [52].

```
repeat $\lceil \log n \rceil$ times
    for each element k in the list in parallel do
        if $s(k) \neq s(s(k))$ then
            $distance(k) \leftarrow distance(k) + distance(s(k))$
            $s(k) \leftarrow s(s(k))$
```

At the end of the computation $distance(k)$ gives the rank of the element k in the list. By associating a processor with each element of the list we can solve the list ranking problem in $O(\log n)$ time. However, the algorithm is not optimal, since the work of the algorithm is $O(n \log n)$ and the sequential time to rank the list is $O(n)$. In attempting to get an optimal algorithm for this problem using *Brent scheduling technique* as in the prefix computation we run into a problem. Because the elements are not initially indexed (as in an array) we can not assign processors to begin subcomputation at defined positions on the list.

A substantial amount of effort has been put into finding a optimal algorithm for the list ranking. The key step in one optimal algorithm is to cleverly *splice out elements from the list so that $O(n/\log n)$ elements remain* in $O(\log n)$ time with $n/\log n$ processors [26]. Then we can solve the list ranking on the reduced list as described above taking $O(\log n)$ time using $n/\log n$ processors. The original list then reconstructed by reinserting the elements that were spliced out. This step can also be done in $O(\log n)$ time with $n/\log n$ processors. The total work of this algorithm is $O(n)$ which is the best sequential time to rank the list.

Wyllie conjectured that it is impossible to find an optimal parallel algorithm for this problem [169]. Cole and Vishkin [26] were able to invalidate Wyllie's conjecture by describing all details of the above algorithm which runs on an an EREW PRAM. The drawback to their algorithm is that it is complicated and has very large constant factors, and they rely on an expander graph construction to solve a scheduling problem that arises. Anderson and Miller [8] gave an another optimal algorithm that runs in $O(\log n)$ time and uses $n/\log n$ processors on an EREW PRAM, which is much simpler and has reasonable constant factors. Moreover, this algorithm does not rely on an expander graph construction, although it is still fairly intricate for practical purposes.

2.5 Graph algorithms

Graphs play an important role in solving real-world problems. Specifically they play a major role in important problems in combinatorial optimisation. For example, in graph colouring we assign colours to a graph such that no two adjacent edges or vertices have the same colour. Edge colouring and vertex colouring can be used to solve time-tabling problems. Another important graph problem is to decide whether a given graph is planar. For example, in the layout of printed circuits one is interested in knowing if a particular electrical network is planar. Other important problems are concerned with the so-called connectivity of the graph in question. A graph is said to be connected if there is a path between any two vertices. A graph is k-vertex (or edge) connected if k is the minimum number of vertices (or edges) whose removal will disconnect the graph. If we think of a graph as representing a communication network, the vertex connectivity (or edge connectivity) becomes the smallest number of communication stations (or communication links) whose breakdown would jeopardise communication in the system. The higher the vertex connectivity and edge connectivity, the more reliable the network.

The design of efficient parallel algorithms for graph problems has presented a challenge since traditional sequential graph search techniques have proved not readily to admit parallelisation. Sequential optimal algorithms for many graph problems commonly use one of two methods to search a graph: *depth-first search* (dfs) or *breadth-first search* (bfs) [49]. At present, neither of these methods has an efficient parallel algorithm, and the most useful of these methods (dfs) is *P-complete*. Thus new tools are needed to replace dfs or bfs. One such tool is the process of *ear decomposition search* (described in section 2.5.3). We need to avoid dfs or bfs in the parallel algorithm. For example, computing connected components is often considered a basic problem and the best sequential algorithm for this problem uses dfs. Two nodes of a graph are in the same component if there is a path from one to another in the graph. An efficient parallel algorithm for connected components on a

CRCW PRAM was described in [70, 73, 144] and the algorithm avoids dfs. Another example is concerned with finding a topological ordering of directed acyclic graph, i.e. assigning a number to each of the vertices such that there is no path from a vertex to lower numbered one. This can easily be done in linear time sequentially, but the algorithm does not obviously lend itself to parallelism. Kucera [85] described an NC algorithm for this problem using the transitive closure technique (as explained in section 2.5.4).

The following sections briefly describe algorithmic techniques which can be used as building blocks for graph algorithms. These exemplify new paradigms for parallel algorithm design.

2.5.1 Euler tour on trees

The construction of a rooted spanning tree, and the computation of various tree functions (for example, preorder and postorder numbering of vertices in the tree, distances of each vertex from the root of the tree, and the number of descendants of each vertex in the tree) are common features in many efficient parallel graph algorithms. It is often the case for particular algorithms that polylogarithmic efficiency is obtained simply because the algorithm has been contrived to perform certain functions on a tree. These functions can be computed by finding a so-called Euler tour of the tree and performing list ranking on the Euler tour. The Euler tour of a tree reduces the computation of many tree problems to some form of list ranking. The Euler tour technique was introduced by Tarjan and Vishkin [152].

An Eulerian circuit is a circuit in a graph which traverses every edge precisely once.

Given an undirected and unrooted tree, by replacing each edge of the tree by two anti-parallel directed edges an Eulerian circuit (or Euler tour) can be constructed optimally by a clever use of the adjacency lists of the tree vertices and the optimal list ranking algorithm (see for example [52], pages 21-24). If we break the Eulerian tour at an arbitrary edge, fixing some edge (i, j) as a first edge of the list so formed, then it is easy to see that the tour represents a depth-first traversal of the tree with i as the root. Tarjan and Vishkin called such a list a *traversal list* of the tree. The parent-child relation, preorder and postorder numbering, number of descendants of each vertex can be determined by ranking the *traversal list* using appropriate weights [152]. Hence, all these tree functions can be computed in $O(\log n)$ time using $O(n/\log n)$ processors on an EREW PRAM.

We can also use the traversal list to compute the distance of each vertex from the root of the tree. This proved to be key subroutine used to compute the biconnected components of a graph [152]. A graph is biconnected if there is no vertex whose removal leaves the graph disconnected. Using the Euler tour in trees, Schieber and Vishkin [140] solved another problem on trees, that of finding the lowest common ancestor of each pair of vertices. This was then used as part of optimal algorithms for computing strong orientations of sparse graphs. Given an undirected graph, the strong orientation problem is to assign a direction to each edge so that the resulting graph is strongly connected. A graph is strongly connected if for any pair of vertices u and v, there exist directed paths from u to v and from v to u.

2.5.2 Tree contraction

Tree contraction is an efficient parallel method of evaluating an expression given the associated tree. The method transforms the input tree in stages using local operations in such a way that an n-node tree is contracted into a single node by local contractions in $O(\log n)$ stages, each of which takes constant time on a PRAM. Optimal algorithms for tree contraction are described by Gibbons and Rytter [53] and others in [29, 47], that run in $O(\log n)$ time on an EREW PRAM.

In addition to expression evaluation, tree contraction has been applied to a wide variety of problems. The technique easily generalises to arbitrary (nonbinary) trees, and has been used to drive parallel algorithms for various graph-theoretic computations on trees such as maximum matching, minimum vertex cover and maximum independent set [62]. Other applications of tree contraction can be found in [105].

2.5.3 Ear decomposition

An *Ear decomposition* of a graph is an ordered collection simple paths called *ears*, such that the end points of each ear appear in previous ears but such that the interior vertices of each ear appear for the first time in that ear. *Ear decomposition search* has been developed for undirected graphs and was suggested as a replacement for dfs to search undirected graphs. This method provides efficient parallel algorithms for several graph problems on the PRAM model. This can be seen in figure 2.2. Several of these parallel algorithms convert to entirely new and optimal sequential algorithms. This is an example of a new emerging discipline enriching an existing one. Surveys of these results can be found in [42, 76, 128, 166].

2.5.4 Matrix computations

Matrix computation provides a fundamental tool for placing many graph problems in NC, using the strategy of repeated matrix multiplication. Let $A = (a_{ij})$ and $B = (b_{ij})$ be $n \times n$ boolean matrices. Let $C = AB$ be the product of A and B; that is, the (i, j) entry of C is defined by $c_{ij} = \oplus_{k=0}^{n}(a_{ik} \otimes b_{kj})$, where $\oplus$ and $\otimes$ are two binary operators. This can be done in $O(\log n)$ time using $n^{2.376}$ processors on a CRCW PRAM [35]. However, the algorithm is of theoretical interest only because it quite complicated and the big-oh notation hides a large constant factor in the running time, and the algebraic structure with the binary operates $\oplus$ and $\otimes$ requires to be a *ring*. The standard method of multiplication in $O(\log n)$ time with n^3 processors on a EREW PRAM, still remains the algorithm of practical choice.

The transitive closure of A (denoted by A^*) is $\oplus_{k=0}^{\infty} A^k$, where $A^0 = I$ (identity matrix) and $A^k = A^{k-1} \otimes A$ for $k \geq 1$. Let matrix D be the matrix $I \oplus A$. It can be shown that $A^* = D^{2\lceil \log_2 n \rceil}$ [68]. Hence, the straightforward method of computing the transitive closure of an $n \times n$ matrix is to compute the $2\lceil \log_2 n \rceil^{th}$ power of the matrix D using repeated squaring as we explained in section 2.3.2.

We can solve several (directed) graph problems by taking powers of the *adjacency matrix* as in the transitive closure problem. For a given graph $G(V, E)$ of n vertices, the adjacency matrix of the graph is the $n \times n$ matrix, $M = (m_{ij})$ (say) such that

$$m_{ij} = \begin{cases} 1 & \text{if edge } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

The transitive closure of a graph $G(V, E)$ is the graph (denoted by G^*) with nodes V and edges $E^* = \{(i,j) \mid \text{there is path from } i \text{ to } j \text{ in } G\}$. Let M^* be the adjacency matrix of G^*. To find M^* it is sufficient to compute the transitive closure of M here $\oplus$ and $\otimes$ are replaced by logical operators **or** and **and** respectively. Hence, by this computation we can find whether one vertex is reachable from another in a directed graph.

As a second example consider finding the shortest path between each pair of vertices in the weighted graph. Here the operators $\oplus$ and $\otimes$ are replaced by **min** and + respectively [85], and the input matrix is the edge-weight matrix (i.e. m_{ij} is weight of the edge (i,j)).

Several other problems on directed graphs can be solved using this strategy of repeated matrix multiplication. These include topological sorting and strong components [128].

Chapter 3

Parallel Approximation Algorithm

A major part of this chapter is the description of an efficient parallel approximation algorithm for finding minimum weight perfect matching in graphs. This represents joint work with A.M. Gibbons and N. W. Holloway, which was published in the proceedings of the 19th *International Workshop on Graph Theoretic Concepts in Computer Science* [136]. Preliminary sections of the chapter provide essential background material.

3.1 Introduction

In sequential complexity theory, it has been the consensus view that the so-called *NP-complete* problems (which includes literally hundreds of computationally important problems, many in the area of combinatorial optimisation) are computationally intractable [3, 46, 49]. Although there is no proof of this fact, so much effort has been fruitlessly expended in the search for polynomial time algorithms that most theoreticians now believe that none exist. The absence of polynomial time algorithms for the *NP-complete* problems has spawned the development of many approximation algorithms which run in polynomial time but which provide an approximation (within guaranteed bounds) to the required result [46]. In a similar vein, NC approximation algorithms for *P-complete* problems have been developed recently [102, 141].

For many problems in P, not much is known concerning their parallel complexity. For example, although the problem of constructing a *minimum weight perfect matching* is known to be in RNC it is not known if it belongs to NC. A problem is said to belong to the class RNC if the problem can be solved in polylogarithmic parallel time by a randomised parallel algorithm, using a polynomially bounded number of processors. A matching in a graph is a set of edges M, so that no two elements of M have a common vertex. If every vertex of a graph is an end-point of some element of M then M is a *perfect matching*. Not every graph contains a perfect matching. Given a weighted graph, a *minimum-weight perfect matching* is a perfect matching whose sum of edge weights is a minimum.

In this chapter an NC approximation algorithm is described for finding a minimum-weight perfect matchings in complete weighted graphs satisfying the *triangle inequality*. In a graph satisfying the triangle inequality, the weight of any single edge forming a triangle with two other edges is less than or equal to the sum of the weights of these other two edges. Such an inequality is satisfied in many natural problems. The problem that we address is an important sub-task for many problems of combinatorial optimisation and features, for example, in solutions to Chinese Postman

Problems (CPP) and in Approximations to the Traveling Salesman Problem (TSP) [49]. Given a graph (directed or undirected) and a weight for each edge, the CPP is the problem of finding a circuit of minimum total weight which contains each edge at least once. The TSP requires to find a minimum weight circuit of a weighted graph which visits every vertex at least once.

3.2 Performance ratios of approximation algorithms

We can measure the performance of an approximation algorithm by comparing the optimal solution and the (approximate) solution produced by the approximation algorithm [46]. If Q is an optimisation problem and I a particular instance of that problem, then the performance ratio of an approximation algorithm A on I is given by $R_A(I)$ defined as follows:

$$R_A(I) = \begin{cases} \frac{A(I)}{Op(I)} & \text{if } Q \text{ is a minimisation problem} \\ \frac{Op(I)}{A(I)} & \text{if } Q \text{ is a maximisation problem} \end{cases}$$

Here $A(I)$ and $Op(I)$ are respectively the approximate solution produced by the algorithm A on I and the optimal solution for I. It is clear from this definition that $R_A(I) \geq 1$ always. However, a useful performance ratio is a ratio that is known never to exceed some constrained value for any instance of the problem. Let R_A denotes the worst-case performance ratio, for all problem instances. Then generally we require to prove that R_A is always constrained for any input.

3.3 Minimum weight perfect matching

As [88] has emphasised, matching problems have played an important rôle in the foundations of sequential algorithmic complexity theory. This is because they are important problems that arise in many guises that can be solved in polynomial time, but for which all naive algorithms take exponential time. In fact, it was in Edmonds' celebrated paper [41] on matching algorithms that the connection between *tractable problems* and *polynomial time solvable problems* was first made. It is likely that matching problems will play a similar rôle in the development of parallel algorithmic complexity theory.

There are practically no extant algorithms placing matching problems in NC with the notable exception of the *maximal matching problem* [67] and certain algorithms for special classes of graphs. The class of problems which can be solved in poly-logarithmic *expected* time using a polynomial number of processors is called RNC (see, for example, [148]and section 2.5.5. of [88]). Most matching problems can be solved in parallel using randomness [77, 110] and so belong to the class RNC. It has been stated [88] that whether a modern definition of a *tractable* problem in parallel computation is one can that can be solved rapidly with randomisation or one that can be solved rapidly without randomisation may ultimately depend upon whether fast parallel algorithms for matching require randomisation.

Even with restrictions on the graph such as completeness and triangle inequality satisfaction, the problem seems very difficult to place in NC. We have therefore addressed the problem of finding an NC *approximation algorithm*. Specifically, we describe an NC approximation algorithm for the minimum-weight perfect matching problem for graphs satisfying triangle inequality for which $R_A = 2\log_3 n$. This is the first such deterministic algorithm with a sub-linear performance ratio. Previously, it was known (see [148]) that there is an NC approximation algorithm for the maximum- (equivalently, minimum-) weight perfect matching problem such that $R_A = n$.

Karp, Upfal and Wigderson [77] described an RNC algorithm for the minimum weight perfect matching problem, which runs in $O(\log n \log^2(Wn))$ time (after the improvements of [45]) using $O(Wn^{3.5})$ processors, where W is the maximum weight of any edge. A faster RNC algorithm was obtained in [110] which runs $O(\log^2 n)$ time using $O(mWn^{3.5})$ processors, where m is the number of edges. These algorithms are in RNC only if W is relatively small (that is, W = $n^{O(1)}$).

Although finding a NC algorithm for minimum weight perfect matching seems to be hard, there are NC algorithms for finding a perfect matching in special classes of unweighted graphs. Examples are dense graphs [37], bipartite graphs with a bounded permanent [60], complements of transitive oriental graphs [63] and line graphs [111]. However, there is no known deterministic NC algorithm for minimum weight perfect matching for complete graphs. The best known deterministic parallel algorithm for complete graphs runs in O($n^3/p + n^2\log n$) polynomial time using p ($\leq n$) processors [115].

An exact solution for minimum weight perfect matching can be computed in O(n^3) sequential time by the intricate algorithm of Edmonds [41] and this provides a target for the work measure of parallel algorithms. There are sequential approximation algorithms for special graphs [66, 121, 137, 151]. The algorithms of [121] find an approximate minimum-weight perfect matching for graphs satisfying the triangle inequality. However, it is not clear that these algorithms can be effectively parallelised. Even if they could be, they would provide a much more intricate solution to the problem solved in the following section. One of the virtues of the algorithm described here is its simplicity.

3.4 Approximate minimum weight perfect matching in a complete weighted graph

We start by providing an overview of the algorithm whose input is a complete weighted graph $G = (V, E)$ with edge set E and vertex set V, where $\mid V \mid= n$ and n is even. The first part of the algorithm concerns the construction of a graph $F = (V, E')$ where $E' \subset E$, thus F may be obtained from E by a set of edge deletions. The essential properties of F will be that each component contains an even number of vertices and the total sum of its edge weights will be less than $(2 \log_3 n)M$, where M is the sum of edge weights of a minimum weight perfect matching in G. The second part of the algorithm first constructs, for each component of F, a Hamiltonian circuit which will be even length. The sum of the edge weights over all such circuits is less than $(4 \log_3 n)M$. A perfect matching in G is then obtained by taking alternate edges on each such circuit and such that (of the two possibilities presented by each circuit) the lightest weight possibility is chosen. The weight, M', of the perfect matching constructed in this way satisfies the inequality: $M' \leq 2M \log_3 n$. We now consider the two parts of the algorithm in more detail.

Afterwards we consider precise details of its parallel execution, justify the bounds on the approximation and consider the complexity parameters.

The Algorithm

1. Construction of F.

The construction is performed over at most $\log_3 n$ phases. At the beginning of each phase, the vertices of G have been partitioned into disjoint subsets whose union is V. Each such subset is called a *super-vertex*. If such a super-vertex contains an even number of vertices of G, then it is an *even* super-vertex, otherwise it is an *odd* super-vertex. Before the first phase, every vertex of G is a super-vertex.

Now, the action of each phase is as follows. In the i^{th} phase, first construct the weighted complete graph G_i which is the graph whose vertex set is the set of super-vertices and the edge (V_j, V_k) between super-vertices V_j and V_k has weight equal to the weight of the lightest edge in G that connects a vertex in V_j to a vertex in V_k. Note that G_i does not hold the triangle inequality. Now construct the complete weighted graph G'_i from G_i as follows. The vertices of G'_i are the odd super-vertices of G_i and the weight of the edge (V'_j, V'_k) between the super-vertices V'_j and V'_k of G'_i is the weight (that is, the sum of the weights of the edges) of a shortest path between these super-vertices in G_i. We now construct a weighted *digraph* G''_i, whose underlying graph (that is, the graph obtained by removing the edge orientations) is a subgraph of G'_i. The vertex set of G''_i is the vertex set of G'_i. We choose precisely one edge to be directed *from* each vertex of G''_i. If e is an edge of least weight with such a vertex as an end-point in G'_i, then the edge chosen from the vertex in G''_i is directed towards the vertex corresponding to the other end-point of e in G'_i. This directed edge has weight equal to the weight of e. Each such directed edge of G_i'' corresponds to a path and so to a set of subset of edges (the edges on the path) of G_i and, for all such directed edges, we now add to F these corresponding subsets of edges. As we shall see later (Lemma 3.4.1), the sum of weights of these edges does not exceed $2M \log_3 n$. To complete the description of what happens within each phase, it remains to say how the super-vertices are constructed for the next phase. Those super-vertices belonging to the same component of G_i'' are coalesced into larger super-vertices. Provided all the super-vertices are not even super-vertices, we enter another phase of the construction of F.

2. Construction of the matching from F

The input to this stage of the algorithm is the graph F. We are interested in the partition of the vertices of G that is implied by the components of F. Each component F_i of F has an even number of vertices of G. We find a minimum weight spanning forest of F, that is, a minimum weight spanning tree T_i for each F_i. For each T_i we then find a preorder numbering of the vertices. Now, for each T_i, such a numbering defines an even-length circuit in G obtained by visiting the vertices in the order of their pre-order indices. For each such circuit, we take alternative edges (of the two possible subsets that can be chosen in this way, we choose that of the smallest weight) to be edges of the approximate minimum-weight perfect matching. The total weight (Lemma 3.4.2) of edges chosen to belong to the approximate weight perfect matching is less than $2M \log_3 n$

end of the algorithm

3.4.1 Validity of the approximation ratio claim

We now prove that the total weight of the edges of the graph F is less than $2M \log_3 n$. First notice that within each iterated phase employed in the construction of F, each odd super-vertex is coalesced with at least one other so that the number of odd super-vertices reduces by a factor of at least two within each phase. However, we continue to iterate only if there are odd super-vertices remaining and this only happens if at least three odd super-vertices are coalesced. Thus $\log_3 n$ repetitions are sufficient to remove all the odd super-vertices (notice that by an elementary theorem of graph theory, there will always be an even number of odd super-vertices). All we now need for our proof is the following Lemma.

Lemma 3.4.1 *The sum of the edge weights of the edges added to F in each iterated phase of its construction is less than or equal to* $2M$.

Proof In any of the iterated phases used in the construction of F, the edges added to F are those belonging, for every odd super-vertex, to a shortest path from such a super-vertex to another. We first show that, in G_t, there exists a path from any odd super-vertex to some other odd super-vertex only using edges of a minimum-weight perfect matching. Consider then any odd super-vertex V of G_t. Now because there are an odd number of vertices of G in V, not all these vertices can be matched by edges of a minimum-weight perfect matching of G which connect pairs of vertices contained in V. There must therefore be an edge of a minimum-weight perfect matching connecting V to some other super-vertex that is a vertex of G_t. This is the first edge of the path whose existence we wish to prove. If this edge takes us to a vertex corresponding to an odd super-vertex then we have finished. If it takes us to a even super-vertex, then it will match one of its constituent vertices and there will be an odd number remaining to be matched by a minimum-weight perfect matching. The implication is that there is another edge from this vertex corresponding to an even super-vertex which takes us on to yet another vertex of G_t. Continuing in this way, we see that there must exist a path from every vertex of G_t corresponding to an odd super-vertex to a similar vertex and that in G_t such a path only uses edges of a minimum-weight perfect matching. Notice, of course, that any two such paths may have edges in common. Let S_k be the sub-set of edges defining the shortest path from vertex k (which corresponds to some odd super-vertex) to some other similar vertex which uses edges of a minimum-weight perfect matching only and let S'_k be the subset defining the path from vertex k to some odd super-vertex as constructed by the algorithm.

We need to obtain a worst case bound on the weight of the *union* of the S'_k in terms of the weight of the union of the S_k. This is because we already have a worst case bound on the weight of the union of the S_k provided by M, the weight of a minimum-weight perfect matching of G. Notice that for all k, $weight(S'_k) \leq weight(S_k)$ because the algorithm chooses minimum-weight paths; however, it does not follow that the weight of the *union* of the S'_k will be less than the weight of the *union* of the S_k because there may be an entirely different sharing of edges between paths in the two cases. To obtain a worst bound, we need to consider the cases in which the S'_k have a minimum union and the S_k have a maximum union. By maximum (minimum) union we mean that the number of shared edges of paths between the odd super-vertices is maximum(minimum). In the latter case, notice that no two of the S_k may share edges whose combined weight is more than half the weight of the lightest set of the two, otherwise the S_k would not be shortest paths of their type. The situation of a maximum weight union of the S_k corresponds to every path sharing half its weight with every other path. The case of minimising the weight of the union of edges of the S'_k is essentially that of practically no sharing (although the detail is a little more subtle, this observation suffices to achieve the bound we seek). Thus, the weight of the union of the S'_k is bounded, in the worst case, by twice the weight the union of the S_k, but the weight of the union of the S_k is bounded by M and so the lemma follows. □

We have proved that the sum of the weights of the edges of F is bounded by $2M \log_3 n$. The following Lemma provides a similar bound on the approximation ratio of the algorithm.

Lemma 3.4.2 *The weight, M', of the perfect matching found by the algorithm is bounded as follows:*

$$M' \leq 2M \log_3 n$$

where, M is the weight of a minimum-weight perfect matching and n is the number of nodes of G. The input G is a complete weighted graph satisfying the triangle inequality.

Proof The total weight of the edges of the graph F is, by Lemma 3.4.1, bounded by $2M \log_3 n$. For each component F_i of F, the total weight of the edges of T_i is less than the total weight of the edges of F_i, because T_i is a minimum weight spanning tree of F_i. Hence, the total weight of all the trees edges is less than $2M \log_3 n$.

For each T_i, consider the standard *twice-around-the-spanning-tree* circuit (see, for example, [49]), C_i, obtained by visiting the nodes in pre-order and making short-cuts to avoid re-visiting nodes that have already been visited. Such short-cuts are always possible because G is complete and they will be *short*-cuts because we have triangle-inequality holding. Thus, for each i, the weight of C_i is bounded by twice the weight of T_i's edges. However, the weight of the edges chosen for the matching from C_i constitute at most half the weight of the circuit and so at most weight of T_i. Thus, over all such circuits, we choose a weight of edges for the matching which is less than or equal to the weight of F and the lemma is proved. □

3.4.2 Parallel execution and complexity of the algorithm

Consider first the construction of the graph F. There are $\log_3 n$ phases and within each, the activities dominating the computation time are the construction of all shortest paths and the coalescing of super-vertices which can be achieved by an algorithm for finding connected components of a graph. As we cite later, there are well known polylogarithmic time parallel algorithms performing these tasks using a polynomial numbers of processors. Other tasks are trivially solved by more efficient parallel algorithms. Thus, we may express the time-complexity for constructing F as $O((SP(n) + CC(n)) \log n)$ using $max(p_{SP(n)} + p_{CC(n)})$ processors, where $SP(n)$ is the parallel time for the all shortest paths problem using $p_{SP(n)}$ and $CC(n)$ is the parallel time for the connected components problem using $p_{CC(n)}$ processors.

Now consider the construction of the approximate minimum-weight perfect matching from the graph F. The dominating activity from the point of view of the computation time might seem to be that of finding a spanning forest. This can be done in $\log^2 n$ time using $n^2/\log^2 n$ processors [25]. A pre-order numbering of tree vertices can be found by the Euler tour technique of [152], and the problem of choosing a set of alternate edges of least weight from a circuit can be computed by employing ranking and summing. Note that the parallel time and work required for the construction of the approximate minimum-weight perfect matching from F are small compared with what are required for the construction of F.

Thus overall, we see that the problem of finding an approximate minimum-weight perfect matching has a parallel solution taking $O((SP(n) + CC(n))\log n)$ time using $max(p_{SP(n)}, p_{CC(n)})$ processors. We can now see what this means in terms of variants of the P-RAM and using the best extant parallel algorithms for the problems of finding all shortest paths and connected components.

Consider first the best practical (in terms of modest constants hidden by the order notation) extant solutions for the all pairs shortest path problem. The problem can be solved in $O(\log^2 n)$ time using $n^3/\log n$ processors on a CREW P-RAM or on an EREW P-RAM. These algorithms are adaptations (see, for example, [52]) of a *common* CRCW P-RAM algorithm of Kucera [85]. Clearly, the same algorithm will run within the same complexity bounds on an *arbitrary* CRCW P-RAM. On the model for which it was described, Kucera's algorithm runs in $O(\log n)$ time using n^4 processors. Now consider the best extant solutions for the connected components problem. The problem can be solved on an *arbitrary* CRCW P-RAM in $O(\log n)$ time using $(m + n)$ processors [144]. For the EREW P-RAM [73] (and therefore for the CREW P-RAM, although an earlier algorithm [70] already existed for this model) the problem can be solved in $O(\log^{3/2} n)$ time using $O(m + n)$ processors, where m is the number of edges.

From the preceding information, we have the following theorem.

Theorem 3.4.1 *There is an algorithm to find an approximate minimum-weight perfect matching of a complete weighted graph satisfying the triangle inequality, with n nodes, having a performance ratio* $R_A = 2 \log_3 n$ *which:*

1. *runs in* $O(\log^2 n)$ *time using* n^4 *processors on an arbitrary CRCW P-RAM.*
2. *runs in* $O(\log^3 n)$ *time using* $n^3/\log n$ *processors on either a CREW P-RAM or an EREW P-RAM.*

For all P-RAM models, the problem of finding all shortest paths (both in terms of time complexity and work) dominates the computation time and the work measure (that is, the processor number, computation time product). The best sequential time-complexity for an exact solution on complete graphs, $O(n^3)$, can still be achieved by the primal-dual algorithm of Edmonds [40] as improved by Gabow [44] and Lawler [87]. Thus, our algorithm is within a factor of $\log^2 n$ of the work measure of Edmonds' algorithm. The faster computation afforded by implementation on the CRCW P-RAM comes (as in commonly the case for P-RAM implementations) at a high cost in terms of numbers of processors required.

3.5 Further work

There are many open problems in the area of parallel approximation. In general, the question of approximating certain intransigent problems in NC, although often difficult does provide some hope for fast parallel computation which may not otherwise exist. We believe that work in the area of parallel approximation is bound to play an important role in a deep understanding of parallel computation. The following describe some further work in this area.

1. **The existence of parallel algorithms**. Some *P-complete* problems do not have an approximating solution in NC for any value of the performance ratio unless $NC = P$ [81, 82, 141]. One such problem is *Linear Programming* (LP) [141]. Given an integer $n \times d$ matrix A, an integer $n \times 1$ vector b, and an integer $1 \times d$ vector c, LP problem is to find a rational $d \times 1$ vector x such that $Ax \leq b$ and Cx is maximised. Identifying such problems will further refine the complexity classes.

2. **Threshold behavior**. Some *P-complete* problems exhibit threshold behavior. For certain values of the performance ratio, an approximation algorithm exists while no such algorithm can exist for other values [9, 142, 141]. For example consider the High Degree Subgraph (HDS) problem, defined as follows: For given a graph G compute the largest d such that the nodes of the induced subgraph of G have degree at least d. This problem cannot be approximated in NC by a performance ratio $R_A < 2$ unless $P = NC$, but it can be approximated to within any R_A for $R_A > 2$ by an algorithm in NC [9]. Many

more examples of this kind are likely to be discovered.

Chapter 4

Realistic Issues

4.1 Introduction

The PRAM is a virtual design-space for a parallel computer, that is the PRAM is a theoretical model of an idealised parallel computer. The PRAM assumes constant-length data paths from every processor to every memory cell. In current technology, this quickly becomes physically unrealisable as we scale up the number of processors and the size of the shared memory. In feasible large scale parallel computers, packing constraints such as this force the inevitability of employing communication networks. A feasible parallel model consists of processing elements each placed at the node of an inter-connection network. The machine memory is split into modules each of which is also located at a node of the network. That is, an intercommunication does not generally provide constant time access between every processor and every memory location. Further delays can occur due to congestion in the network and contention at the memory modules. The need for such networks may be obviated in the long term by the appearance of new technologies (optical communication provides one such hope [8, 99, 131]), but for the foreseeable future we will have to contend with the complication of networks with communication delay and restricted message passing density. The PRAM does not account for possibility processor failure and break down in communication links. Moreover, the PRAM assumes that it operates in lock step synchrony, but the processors of the feasible models may or may not have to be synchronised. This chapter reviews all of these issues. In chapter 6 we demonstrate that using the facts described in this chapter there is no theoretical hindrance in designing a feasible large scale parallel computer.

4.2 Interconnection Networks

A feasible parallel computer is a number of processing elements interconnected by a network. Each processing element may thought of as a conventional random access machine. The interconnection network can be depicted by a graph in which edges represent communication links and nodes represent processing elements or switching elements. Each processing element or switching element has ability to route messages (one in unit time) to adjacent nodes of the graph. The interconnection networks can be broken into two broad classes depending on whether or not a processing element is located at every node. In a *multistage* or *indirect* network, the computing elements and memory modules are interconnected by a network of switches. In a *direct* network, there is a (processing element) computing element and a memory module at each node of the network. Note that, in both networks each computing element can access any memory module. Hence, we consider the union of the memory modules in both networks as a *virtual shared memory*.

The cost of communication is related to the topology of the graph. To reflect physical packaging constraints we require that the degree of the graph should be small. The degree of a graph is the maximum number of edges attached to any node of the graph. The diameter of a graph is the maximum minimum-length path between any pair of nodes of the graph. Two processors may be separated by a path of this length. Therefore the diameter provides a lower bound on communication delay in the graph. Thus we require that both the degree and the diameter of a graph are small (preferably constant or growing slowly, for example logarithmically with the size of the graph). Note that we should consider the tradeoff between diameter and degree. For example, Moore graphs, which are graphs of minimum diameter for fixed network size and degree, have been studied in [69]. However, the need to able to support high parallel message passing density dictate a little away from optimality in Moore sense. Also we may advantageously use recursively decomposable graphs which not only naturally support recursive algorithms but can also aid physical construction and size enhancement. Here we describe some of the networks that have been proposed for general purposes [88, 124, 146, 147, 153].

4.2.1 The hypercube family

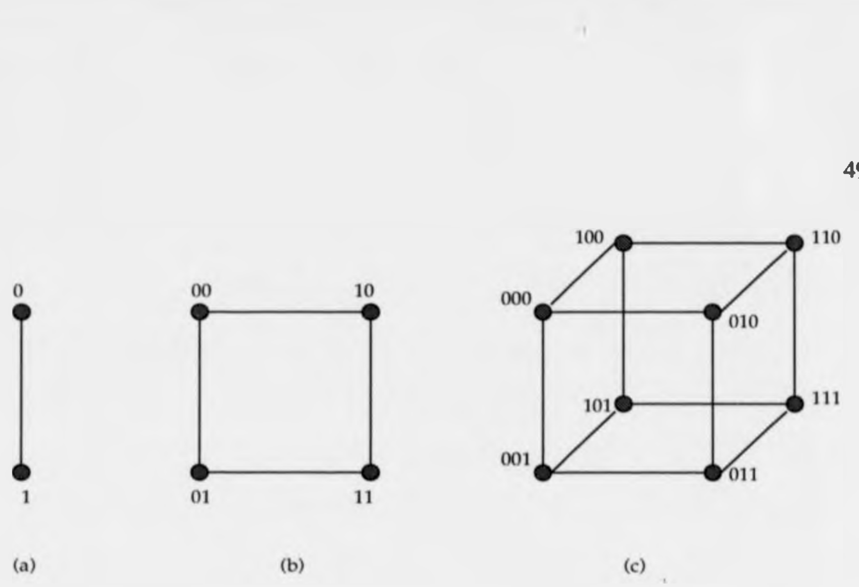


Figure 4.1:

Hypercube

The d-dimensional hypercube, *d-hypercube*, has $n = 2^d$ nodes and $d2^{d-1}$ edges. For example, 1, 2 and 3 dimensional hypercubes are shown in figure 4.1 (a), (b) and (c) respectively. Nodes are addressed by binary strings of length d and edges connect binary strings which differ in precisely one bit position. As a consequence, each node is incident to $d = \log n$ other nodes. Thus the degree of the *d-hypercube* is $\log n$. The length of a shortest path between any pair of nodes is the number of positions in which their binary strings differ. Since binary strings of any two nodes differ in at most d positions, the diameter of the hypercube is d. A shortest path from a node, A, to a node, B, can be constructed by successively visiting the nodes whose labels are those obtained by modifying the bits of A one by one (for example, from left most significant bit to right most significant bit) in order to transform A into B. For example, the shortest path between 000 and 011 in the *3-hypercube* is $000 \rightarrow 010 \rightarrow 011$.

Edges which connect nodes which differ in the i^{th} position are called edges of the

i^{th} dimension. The hypercube is a recursively decomposable graph. If we remove the i^{th} dimension edges of the *d-hypercube*, for $1 \leq i \leq d$, then we get two disjoint copies of a *(d-1)-hypercube*. Conversely, a *d-hypercube* can be constructed from two *(d-1)-hypercubes*, by joining every vertex of the first *(d-1)-hypercube* to the vertex of the second having the same number (or binary string). Indeed, it suffices to renumber the nodes of the first cube as $b_1 b_2 \cdots b_{d-1}0$ (add 0 as last bit) and $b_1 b_2 \cdots b_{d-1}1$ (add 1 as last bit) , where $b_1 b_2 \cdots b_{d-1}$ is a binary string representing the two similar nodes of the *(d-1)-hypercubes*.

The cube-connected cycles, butterfly and Benes networks

Although the hypercube is quite powerful from a computational point of view, the degree of the hypercube grows logarithmically with its size. This is a disadvantage of its use as an interconnection network for parallel computation. The cube-connected cycles [123], the butterfly network and the Benes network [15] can be regarded as constant-degree variations of the hypercube. These networks have properties similar to those of the hypercube.

A d dimensional cube connected cycles (*d-cube connected cycles*) is a *d-hypercube* in which each node of the hypercube has been replaced by cycle of d nodes. Hence the *d-cube connected cycles* has $n = d2^d$ nodes. The i^{th} dimensional edge originally incident to a node of the hypercube is now connected to the i^{th} node of the corresponding cycle. Each node of the cube connected cycles can be represented by a pair (p, c), where p is the position of the node of the cycle c. For each node (p, c) of the *d-cube connected cycles* c is a d-bit binary string and $1 \leq p \leq d$, since there

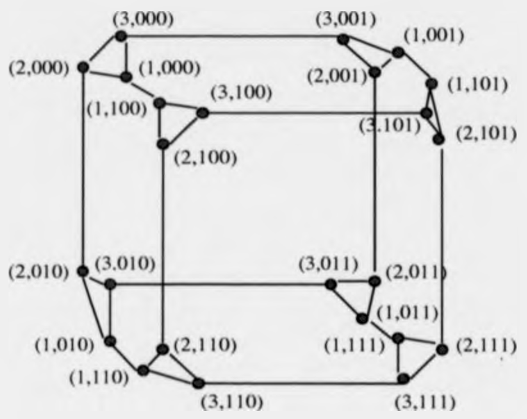


Figure 4.2:

are d nodes in any cycle of the *d-cube connected cycle* and each cycle corresponds to a node of the *d-hypercube*. Two nodes of the *d-cube connected cycles* (p, c) and (p', c') are connected by an edge if and only if $p = p'$ and c differs from c' in precisely the p^{th} bit, or $c = c'$ and $|p - p\prime|$ is 1 or $d - 1$. Figure 4.2 illustrates the graph for $d = 3$. It is easy to see that, for all d, the degree of a *d-cube connected cycles* is three. The diameter is $3\lceil d/2 \rceil$ or $3\lceil \log n/2 \rceil$ for a *d-cube connected cycles*. This is because a message from (p, c) to (p', c') can reach (p, c') within d steps then (p, c') to (p', c') takes at most $\lceil d/2 \rceil$ steps.

Figure 4.3 shows the butterfly of dimension 3. In general, a d dimensional butterfly, *d-butterfly* or *n-input butterfly*, has $2^d(d + 1)$ (or $n(\log n + 1)$) nodes and $2^{d+1}d$ (or $2n \log n$) edges. The butterfly network is an example of a *multistage* network. The nodes are divided into $(d + 1)$ levels with 2^d nodes in each level. Sometimes, the nodes on level 0 are called *inputs* and the nodes on level d are called *outputs*. Let node (r, l) refer to the node on the r^{th} row and the l^{th} level, where r is a d-bit binary

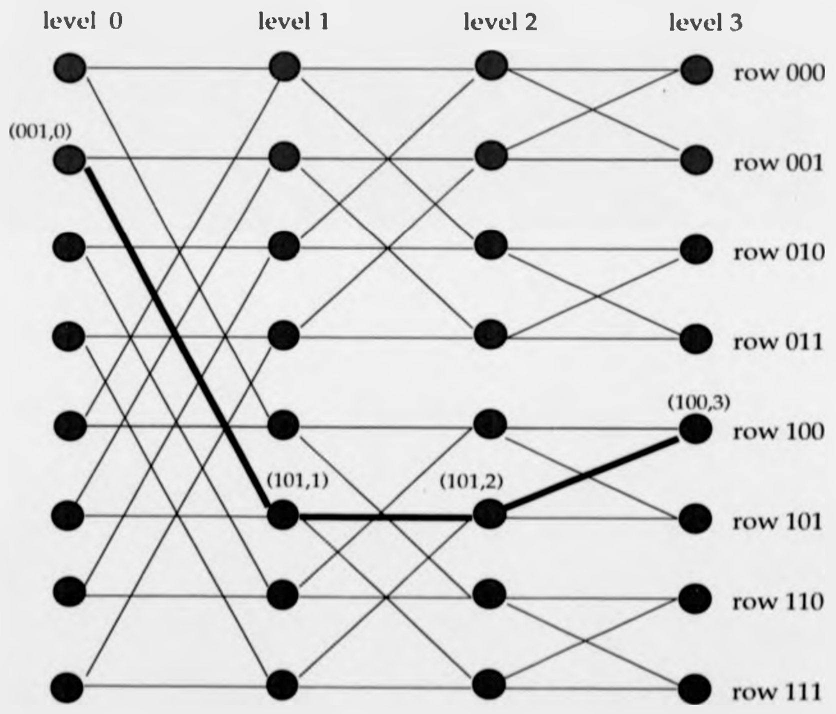

level 0
level 1
level 2
level 3
row 000
row 001
row 010
row 011
row 100
row 101
row 110
row 111
(001,0)
(101,1)
(101,2)
(100,3)

Figure 4.3:

string and $0 \leq l \leq d$. There is an edge between nodes (r, l) and (r', l') if and only if $l = l' + 1$ and either $r = r'$ or r and r' differ in precisely l^{th} bit. The entire network is made of such "butterfly" pattern hence the name. The butterfly has a interesting recursive structure, by removing level 0 nodes of the *d-butterfly* we can obtain two *(d-1)-butterflies*. Moreover, a *d-hypercube* can be obtained from a *d-butterfly* by collapsing each row of nodes, i.e. i^{th} node of the hypercube corresponds to the i^{th} row of the butterfly.

There is a unique path of length $\log n$ (or d) from every *input* node to every *output* node in the *d-butterfly*. This unique path is referred to sometimes as the logical path. The path between input node u and output node v can be constructed by using the edge from a i^{th} level node (w, i) to a $i + 1^{st}$ level node $(w', i + 1), 0 \leq i \leq d - 1$, such that w and w' differ in the $(i + 1)^{st}$ bit if u and v differ in the $(i + 1)^{st}$ bit otherwise $w = w'$. In other words, the path from an input to an output moves downwards at a node on the i^{th} level $0 \leq i \leq (d - 1)$ if the $(i + 1)^{st}$ bit of the destination is a 1, and upward otherwise. The unique path from $(001, 0)$ to $(100, 3)$ is shown in figure 4.3. As a simple consequence of this fact, we can see that any two nodes in the *d-butterfly* can be connected by a path of length at most $2 \log n$ (or $2d$). The diameter of the *d-butterfly* is $2 \log n$ or $2d$, and the (maximum) degree is 4.

A number of variations of the butterfly have been proposed in the literature (for example see [84]) which include the wrapped butterfly, the omega network, the flip network, the Banyan and Delta network, the k-ary butterflies and the Benes network. Among the variations the Benes network has an interesting property. The d dimensional Benes network, *d-Benes network* is constructed from two *d-*

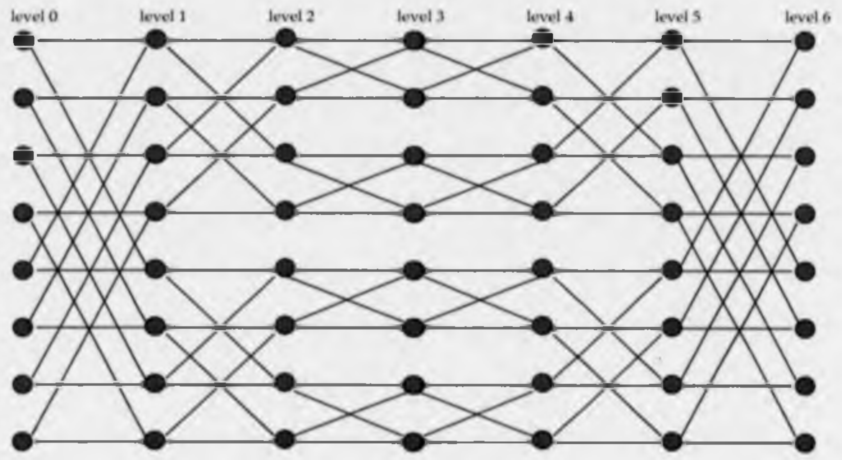


Figure 4.4:

butterflies connected back-to-back. Figure 4.4 shows the 3-Benes network. On the Benes network of d dimension we can construct edge-disjoint paths one path from each node on level 0 to a unique node at level $2d$, for any permutation of the inputs to the outputs [15, 50, 168].

4.2.2 The shuffle-exchange and de Bruijn networks

The d dimensional shuffle-exchange graph, *d-shuffle-exchange* graph, has n nodes and $3n/2$ edges, where $n = 2^d$. Each node i, for $1 \leq i \leq n-1$, is labeled with a corresponding d-bit binary string [150]. There are two types of edges, a ***shuffle edge*** connects any node $b_1b_2..b_m$ to the node $b_2..b_mb_1$ and an ***exchange edge*** connects any node $b_1b_2..b_m$ to the node $b_1b_2..b'_m$, where $b_m \neq b'_m$. Hence the maximum degree of *d-shuffle-exchange* graph is three, for all d. For example, see figure 4.5 for a 3-dimensional shuffle-exchange graph. In the figure dashed edges represent

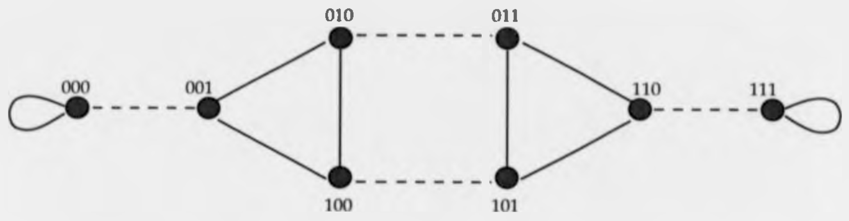


Figure 4.5:

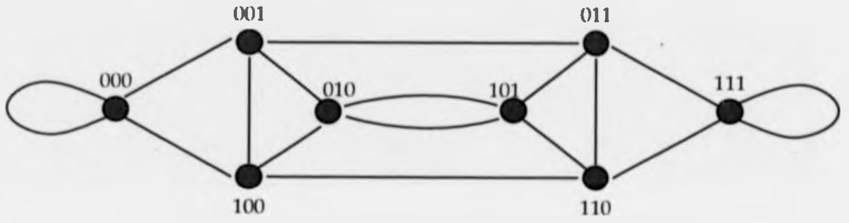
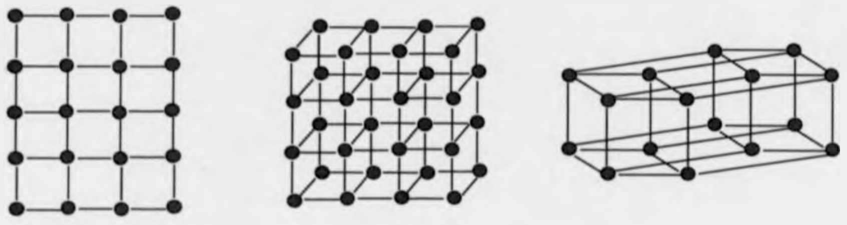


Figure 4.6:

exchange edges, and solid edges represent shuffle edges. The diameter of a *d-shuffle-exchange* graph is $2\log n - 1$ or $2d - 1$. We can construct a path of length at most $2\log n - 1$ between any two nodes in the *d-shuffle-exchange* graph by simply using the exchange edges and shuffle edges alternately.

The d dimensional de Bruijn graph has n nodes, where $n = 2^d$. The nodes are labeled from 0 to $n - 1$ with binary labels. The graph has $2n$ edges. There is an edge between two nodes u and v if and only if $u = b_1b_2\cdots b_d$ and v is $b_2b_3\cdots b_d0$, $b_2b_3\cdots b_d1$, $0b_1b_2\cdots b_{d-1}$ or $1b_1b_2\cdots b_{d-1}$. The d dimensional de Bruijn graphs have diameter of d ($= \log n$) and their degree is four. Figure 4.6 shows the 3-dimensional de Bruijn graph. In chapter 5 we shall look this graph in more detail.



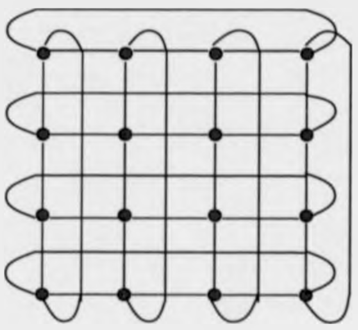
(a) (b) (c)

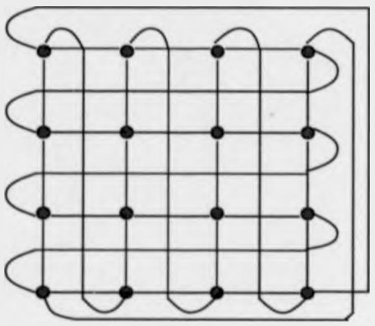
Figure 4.7:

4.2.3 Meshes

Meshes are another class of network that have received wide attention. A d dimensional mesh, *d-mesh*, of dimensions $d_1, d_2, \cdots, d_d$ has nodes $\{1, \cdots, d_1\} \times \{1, \cdots, d_2\} \times \cdots \times \{1, \cdots, d_d\}$ and edges connecting each node of the form $[z_1, z_2, \cdots, z_i, \cdots, z_d]$ with nodes $[z_1, z_2, \cdots, z_i \pm 1, \cdots, z_d]$, for $1 \leq i \leq d$. Such a mesh is called a $d_1 \times d_2 \times \cdots \times d_d$ mesh. The diameter of a *d-mesh* is $(d_1 - 1) + (d_2 - 1) + \cdots + (d_d - 1)$ and it has degree of $2d$, if each $d_i \geq 3$. If all the dimensions are of the same size, N (say), then the diameter is the minimum for fixed number of nodes and d. That is, the number of nodes n is equal to $N \times N \times \cdots \times N = N^d$ and the diameter is $\Theta(n^{1/d})$. Moreover, if all the dimensions equal to 2 then the *d-mesh* is the *d-hypercube*. (4×5) *2-mesh*, $(4 \times 4 \times 2)$ *3-mesh* and $(2 \times 2 \times 2 \times 2)$ *4-mesh* (or *4-hypercube*) are shown in figure 4.7 (a),(b) and (c) respectively.

Although (low dimensional) meshes are relatively simple to construct, they suffer from having large diameter. However, by adding edges to the mesh the diameter can be reduced by a small constant factor. For example, figure 4.8(a) shows a (4×4)





(a) (b)

Figure 4.8:

mesh with *wrap-around* connections and figure 4.8(b) shows the same mesh with *toroidal* connections.

By adding nodes and edges to the d dimensional $N \times N \times \cdots N$ mesh, such that the N nodes of each one-dimensional "*row*" are the leaves of a complete binary tree, we can construct a graph called a ***mesh of trees***. This has the small diameter $2d \log N$. For example, consider a 2-dimensional $N \times N$ mesh of trees, which can be constructed from an $N \times N$ mesh by adding nodes and edges to form a complete binary tree in each row and each column. In a 2-dimensional mesh of trees any two nodes u, v can be connected by a path of length at most $4 \log N$. Suppose u is a node on the i^{th} row and v is a node on the j^{th} column and let w be the node on the i^{th} row and j^{th} column. We first construct a path of length at most $2 \log n$ from u to w using the tree of the i^{th} row and then finish the path in at most $2 \log n$ steps from w to v using the tree of the j^{th} column. Hence the diameter is $4 \log n$.

4.2.4 Randomly-wired networks

Among randomly-wired networks the most popular one is called *multibutterfly*. Informally, the d dimensional *multibutterfly* is like a *d-butterfly* in which the nodes at each level are randomly connected to the next level [155]. The object of random wiring is that the logical path between the input nodes (the nodes on level 0) and the output nodes (the nodes on level d) of the d dimensional *multibutterfly* can be realised by a myriad of physical paths. We shall see in section 4.3.3 that the large number of these paths allow to construct an optimal routing in the network. Recall that, there is the unique path of length d between every input node and every output node in the *d-butterfly*. To define the *multibutterfly* network more formally, we need to understand the graph called a *splitter*, which is the basic building block of the network.

An $(\alpha, \beta, m, 2k)$- or m-input- *splitter* is a bipartite graph $G = (A, B, E)$, where $|A| = |B| = m$. A graph is *bipartite*, if it is possible to partition the vertices of the graph into two subsets, A and B, such that every edge of the graph connects a vertex in A to a vertex in B. The nodes in A are are called *inputs*, and the nodes in B are called *outputs*. The *outputs* are divided into two *blocks* of $n/2$ nodes. The *block* consisting of the first $n/2$ nodes is called the *upper block*, and the block consisting of the remaining nodes is called the *lower block*. The edges from the inputs to the *upper block* are called *up* edges, and those to the *lower block* are called *down* edges. The *up* and *down* edges are chosen at random subject to the constraint that each *input* is incident to k *up* and *down* edges, and each *output* is incident to $2k$ edges. Moreover, the *splitter* has (α, β)-*expansion*, i.e., every subset of *inputs*,

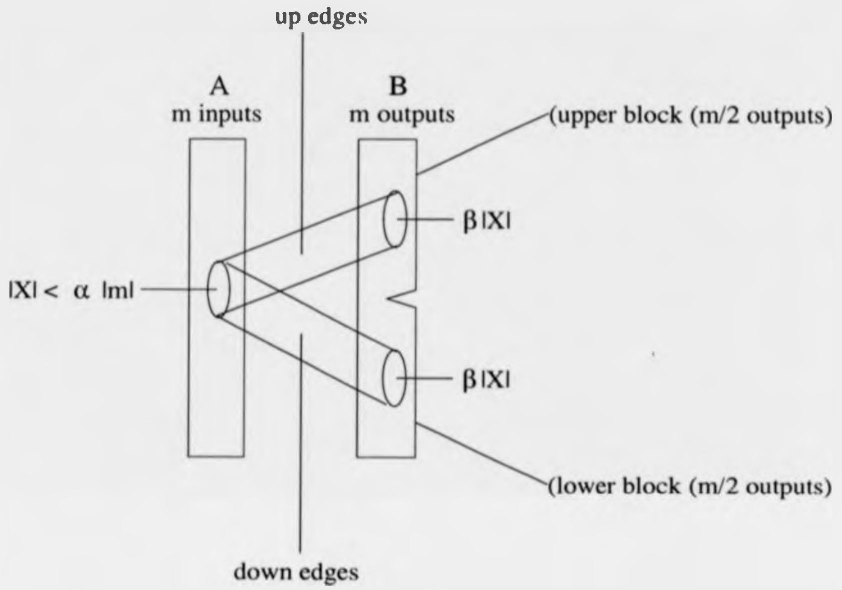


Figure 4.9:

X, $|X| \leq \alpha m$, is connected to at least $\beta|X|$ *outputs* in the *upper block* and $\beta|X|$ *outputs* in the *lower block* (see figure 4.9). Note that, although *splitters* can be constructed deterministically in polynomial time, random choice of edges provides the best known possible expansion [71, 95, 155]. See for example [89] for recent work on practical implementations of this network.

An (n, k, α, β)-*multibutterfly* is a $\log n$ dimensional network, with n input nodes and n output nodes. The *multibutterfly* and the $\log n$-*butterfly* are strongly related to each other. The only difference between the two networks is that the degree of $\log n$-*butterfly* is 4, and the degree of the *multibutterfly* is $2k$. More precisely, in the *multibutterfly*, the edges from level l to level $l+1$ in rows $jn/2^l$ to $(j+1)n/2^l$ form a $(\alpha, \beta, n/2^l, 2k)$- or $n/2^l$-input- *splitter*, for all $0 \leq l \leq \log(n-1)$ and $0 \leq j \leq 2^l$.

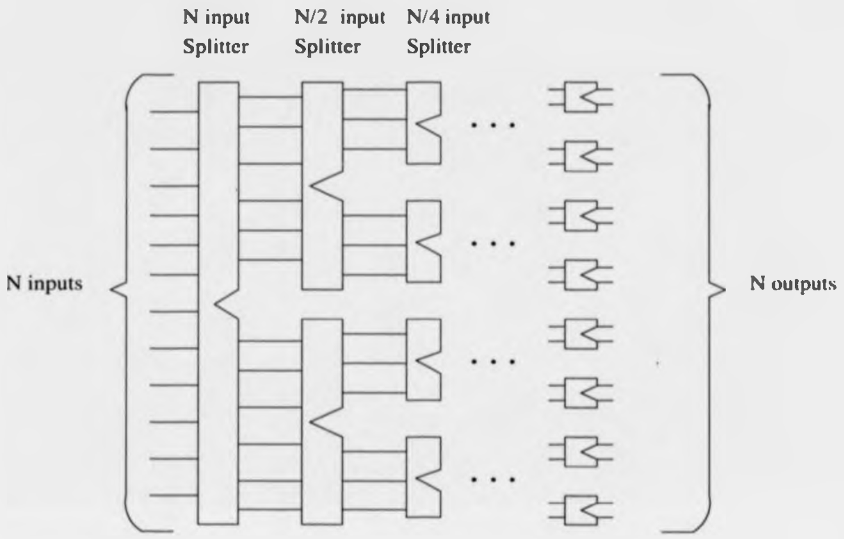


Figure 4.10:

The nodes on each level of *multibutterfly* can be partitioned into *blocks*. All the nodes on level 0 belong to same *block*. On level 1, there are two *blocks*, one consisting of the nodes that are in the upper $n/2$ rows, and other consisting of the nodes that are in the lower $n/2$ rows. In general, there are 2^l *blocks* of size $n/2^l$ on level l, for $0 \leq l \leq \log n$. Each *block* b_l on level l is connected to two blocks of size $n/2^{l+1}$ on level $l+1$. We refer to the two blocks as the *upper block*, u_l, and *lower block*, l_l. Now u_l and l_l contain the nodes on level $l+1$ that are in the same row as the upper $n/2^{l+1}$ nodes of b_l, and the same rows as the lower $n/2^{l+1}$ nodes of b_l, respectively. The three *blocks*, b_l, u_l and l_l, form the splitter $(\alpha, \beta, n/2^l, 2k)$, for which b_l forms *inputs* and $(u_l \cup l_l)$ forms *outputs*. Figure 4.10 shows a *multibutterfly* with n inputs.

Like in the *butterfly*, the logical path between every input nodes and every output nodes can be constructed by using *up* and *down* edges. Furthermore, since each node has k *up* and k *down* edges, each step of a logical path can take any one of k edges. Hence, we have a large number of choices to construct one logical path.

4.3 Contention and congestion

In the PRAM there are conventions for multiple access to any one memory location depending on whether the PRAM is defined as EREW, CREW or CRCW. In feasible machines shared memory is distributed amongst memory modules, and the memory modules are connected by an interconnection network. Each memory module contains many memory locations, and each is capable of serving only a constant number of requests during each time step. This is because, the interconnection network can only carry a certain density of messages, and the memory modules can only store or retrieve a constant number of requests per time unit. *Congestion* occurs when network links or nodes are overloaded with messages to pass. *Contention* occurs when memory modules are overloaded with memory requests in any time step. Local congestion occurs at so-called *hot spots*. Unless hotspots are managed, we might not efficiently implement PRAM algorithms on the feasible models. This is because excessive accesses and transmission will be serialised and served over several sequential time steps.

One solution to avoid contention is to randomly distribute the logical addresses to the memory modules using some *hash functions*. Another solution is to keep the copies of each logical address in several memory modules [104, 156]. A special kind of contention occurs when running a CRCW or CREW PRAM algorithm, several processors attempt to access (read or write) a value in the same memory location.

This can be avoided by a *combining mechanism*. Note that if the patterns of shared memory access are known in advance then we can deterministically distribute the shared memory among the memory modules to avoid contention.

A large amount of work (for example see [89] for a list of references) has been done on the development of efficient routing algorithms for interconnection networks, to reduce congestion and getting the right data to the right place within a reasonable time. A wide variety of routing problems may arise in practice, such as *one-to-one* routing problems or *partial permutations*: at most one message starts at each processing element and is destined for at most one processing element, *one-to-many* routing problem: one message may be destinated for more than one place, *many-to-one* routing problem: many messages are destined for the same place. In general, an *h-relation* is a routing problem in which at most h messages are sent from each processing element and at most h messages arrive at each processing element. The natural lower bound for routing is the diameter of a network. Deterministic parallel routing strategies can often be improved by using randomised algorithms. The routing paths can be set up in two ways. First, in an *off-line* fashion, the routing paths are precomputed using global information. Second, in an *on-line* fashion, the routing paths are made using only local information. We are interested in *on-line* routing, the requirement of global information for *off-line* routing is not often available, or can only be obtained with excessive time-penalties.

4.3.1 Hashing

The most promising method to avoid contention is to randomly hash the memory. The PRAM memory locations are distributed among the memory modules of a feasible machine using some hash function. In this section we consider that concurrent access does not occur (i.e. we consider only EREW PRAM algorithms). Suppose, an EREW PRAM algorithm uses M memory locations. To implement the algorithm with p memory modules we randomly distribute the M memory locations among p memory modules. More precisely, all of the EREW PRAM memory locations which are logical addresses, are mapped into memory locations which are physical addresses of a feasible model, using some randomly chosen hash function h from S_M, h: $[0, \cdots, M-1] \rightarrow [0, \cdots, M-1]$. S_M is the full set of permutations of $[0, \cdots, M-1]$. Let memory module p_z, $0 \leq z \leq p-1$, contain the content of location x, if logical address x mapped into a physical address y, and physical address y is in the memory module p_z, i.e.,

$$\begin{aligned} &h(x) = y \quad \text{and} \\ &y \bmod p = z, \quad x, y \in [0, \cdots, M-1]. \end{aligned}$$

The motivating idea is that if the random mapping is used by some hash function, then we can minimise the maximum number of memory requests which are assigned to the same memory module. Let R_{max} be the maximum of R_z, where R_z is the number of requests for addresses located in module z, $0 \leq z \leq p-1$. The requests for each module are queued in some order and served sequentially, thus we need R_{max} to be small. In addition, we need to minimise the time for computing $h(x)$, $0 \leq x \leq M-1$, and storing the mapping. If h is a totally random function of the form $h : [0, \cdots, M-1] \rightarrow [0, \cdots M-1]$ (i.e. randomly chosen from S_M), then every computing element needs additional, $\log M^M = M \log M$ bits (assuming a suitable encoding) to store an element $h \in S_M$, which can be expensive if M is large. Fortunately, it is sufficient for us to choose h randomly from among a much smaller set, H (say), of easily computable functions. Hash functions for this parallel context have been analysed by Mehlhorn and Vishkin [104]. Their work is based on *universal hashing* as introduced by Carter and Wegman [21].

Suppose an arbitrary step of an EREW PRAM requests a set S up to v number of addresses of the shared memory (i.e. $|S| \leq v$), where $S \subseteq [0, \cdots, M-1]$. Then we need to make sure that $h(S)$ is spread evenly among p memory modules. Consider the class H of hash functions h of the form,

$$h(x) = (a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}) \bmod M,$$

where $a_0, a_1, \cdots, a_{k-1}$ are k integers chosen uniformly and independently at random from the interval $[1, M-1]$. Let $R_{max}(p, v, H)$ be the maximum number of requests to the memory module which has the largest such number, with respect to all possible S addresses of size v, and with respect to all hash functions of H. Using the results of [104] Valiant proved the following results [158].

Theorem 4.3.1 *Let $v = p \log p$ and $k = 4 \log p$. If M is prime then with a high probability $R_{max}(p, v, H) = 3 \log p = 3.v/p$.*

Note that if M is not prime then we can choose M' where M' is the smallest prime larger than M, and define the corresponding H as before but for M' rather than

M. From theorem 4.3.1 we can see that v requests are spread evenly among p memory modules, with high probability each module will get no more than $3.v/p$ requests which is only three times the expected number. However, H is not a class of permutations, the hash functions randomly drawn from H are not one-to-one. Up to $k = 4\log p$ addresses may map to the same memory location. This can be avoided with high probability by choosing different hash functions of the form, $h \ mod \ \lceil M/p \rceil$, where $h \in H$ and $k > 4\log p$ [158]. Another problem of this hashing occurs if $v = p$. Then rather than having a constant number of requests to each memory module, with high probability one memory module will get about $\log p / \log\log p$ number of requests and some will get none. Hence, we need to ensure that $v \geq p\log p$.

Since we are using the class of polynomials of degree $k = 4\log p$, we need special purpose hardware to compute these polynomials in constant time. Note that O($\log\log p$) parallel steps are needed to evaluate the $\log p$ degree polynomials. However, recent analysis shows that a fixed degree polynomial is sufficient [83, 145, 130, 109]. For example, the class of hash functions of the form $((a + bx) \ mod \ M')$, where $a, b \in [1, M' - 1]$, will result in excellent performance, if the choice of a, b is suitably restricted [109]. The following theorem is proved in [83] for the class of hashing functions of fixed degree k.

Theorem 4.3.2 *Let $v = p^3$ and let $k > 6\alpha$. If $M = v^\alpha$ then with high probability $R_{max}(p,v,H) = 2.v/p$.*

Note that whenever more than $2.v/p$ requests are generated for any memory module we rehash the entire memory, using a new randomly chosen hash function $h \in H$.

However, the probability of rehashing is significantly small.

4.3.2 Combining

In the last section we saw how to evenly distribute the memory accesses among memory modules when all of the memory accesses are for distinct memory locations (or addresses). Several or all the computing elements may wish to access not only the same memory module, but also the same memory location at the same time. This may happen when running a CRCW or CREW PRAM algorithm. Here, randomly distributing the PRAM memory to the memory modules cannot help. The number of accesses for the same memory location at the same time will be the same no matter where the memory location physically exists.

Fortunately, we can solve this (*concurrent*) access problem by using a so-called combining technique. For example, in a butterfly network each request for accessing a common memory location moves along the directed path from its source to the destination in the interconnection network. These paths are in general not disjoint. Thus these requests can be viewed as flowing from the leaves of a tree to the root. Then as explained below we can combine these messages into a single message at a node of the tree.

Suppose the requests are for reading the same memory location, concurrent read, then there is no need to send more than one read request along any branch of this tree. Of course, the reply message needs to flow in the reverse direction, along each edge of the tree so that each requesting processor receives a reply. To accomplish this, whenever the read requests are combined at a node of the tree, the sources of the requests in that stage are stored at that node. Suppose several computing elements want to write a value in the same memory location at the same time, concurrent writes (write conflicts can be resolved as explained in section 2.2). Whenever two or more concurrent write requests meet at a node of the tree, the node can send any one of the requests. Moreover, for concurrent write we do not have to worry about a flow of information in the reverse direction. Note that, two or more requests to distinct memory locations within the same memory module might be passed through a node, in which case the requests should be concatenated.

One approach to implementing combining is to use combining networks, i.e. networks that can combine and replicate messages in addition to delivering them in a point-to-point manner. The NYU Ultracomputer incorporates a combining network; computing elements and memory modules are connected by switches with the geometry of the Omega network [58], where the switches are capable of combining the messages. The experiments on the NYU Ultracomputer reveal poor performance, and combining increases the switch size and cost [119]. The Fluent machine of Ranade [129] supports inexpensive hardware for a combining network, and the combining (based on [130]) is efficient. The nodes of the Fluent machine are connected with the geometry of the *butterfly*, where each node contains a computing element, a memory element and 6 switches.

The reason that Ranade's combining works well is that the requests leaving each node are sorted by destination. This works with the help of *ghost messages*. These contain information on the minimum location address, which can identify to which subsequent messages can be sent. After a request leaves a node no more requests to the same destination will arrive at that node. In other words if two requests to a common destination pass through a node then they will pass through the node at the same time, in which case the node combines the two requests and forwards the result. So, a request waits at each node until the node determines that there are no more messages arriving at the node which will request to the same destination, or until another request to the same destination arrives.

Ranade's combining method ensures that the queue-size of the requests at each node is $O(1)$. Moreover, if a memory of CRCW PRAM with size M is randomly distributed among p nodes of the butterfly (each node is a computing element and a memory module), then the combining gives the following theorem. The memory is randomly distributed by the class of hash functions of the form $((a + bx)\ mod\ M')$, where $a, b \in [1, M' - 1]$ and M' is a fixed prime no less than M.

Theorem 4.3.3 *[129] Let p be the number of accesses requested by a CRCW PRAM at any step. The p accesses can be realised in a p node butterfly in time* $15 \log p$ *with high probability.*

In addition, concurrent requests to the same memory location can be provided without the use of combining networks. For example, Valiant [158] and Kruskal *et al.* [83] provide algorithms for simulating concurrent requests on networks. The simulation uses a sorting algorithm of [127] to sort the concurrent requests according to the destination addressees so that access requests to the same location in memory will be adjacent in the sorted array. In [159] Valiant described another algorithm which is more efficient and practical than previous solutions [158, 83].

Theorem 4.3.4 *[159] For any constant $\epsilon > 0$, $v \leq p^{1+\epsilon}$ requests of CRCW PRAM can be realised on a p node network in optimal $O(p^{\epsilon})$ time with high probability.*

The results assume that each node of the network has a computing element and a memory module, and the memory of a CRCW PRAM is distributed among the memory modules using the randomly chosen hash function as previously described. Moreover, the v requests of a CRCW PRAM are spread among p components such that each component sends at most p^{ϵ} requests.

4.3.3 Routing

The simplest kind of routing problem is a *1-relation*. In fact *1-relations* can be realised in a *p-node* network by sorting the p items. If processing elements are indexed according to some natural scheme, then by sorting the items which are distributed across the network, one located at each processing element, we can route the item which would be the i^{th} (in a sorted list of the items) to the processing element with index i.

Among deterministic parallel sorting algorithms the fastest known is based on the *AKS sorting network* which is a *p-node* bounded degree network and capable of sorting p items in $O(\log p)$ time [5]. Although the algorithm is asymptotically optimal the constant factor involved in the run time is large, and the network topology is complicated and nonregular. Batcher's sorting algorithm [13] which has been known for several decades remains the most practical algorithm for sorting. For example, the algorithm can be implemented in $O(p)$ time on a $p \times p$ mesh, and in $O(\log^2 p)$ on a *p-node* hypercube and on a *p-node* shuffle-exchange. This implementation matches the lower bound for the mesh, and is a $\log p$ factor worse than the lower bound for the hypercube and shuffle-exchange networks. Recently, Cyper and Plaxton [36] described a sorting algorithm for sorting p items on a *p-node* hypercube in $O(\log n(\log\log p)^2)$ time or $O(\log p(\log\log p))$ time with a substantial amount of *off-line* computation, which is very close to the lower bound. It is not known whether or not there is an *on-line* deterministic algorithm to realise a *1-relation* or to sort p items on a $\log p$ dimensional hypercubic network in $O(\log p)$ time.

If we restrict a deterministic routing algorithm to be *oblivious*, then the following lower bound holds. A routing algorithm is *oblivious* if the routing paths are determined only by the sources and destinations, and not in any way by the interacting traffic on the network.

Theorem 4.3.5 *[19, 72] For any oblivious algorithm there is a permutation routing problem or 1-relation which will take $\Omega(\sqrt{p}/d^{3/2})$ time on a p-node degree-d network.*

Among *oblivious* algorithms the *greedy* algorithms are known to perform very well for almost all routing problems, and very poorly for some of the most important and most common routing problems in practice [88]. A *Greedy* algorithm routes every packet along a shortest path to its destination. For example, on any $\log p$ dimensional hypercubic network the greedy algorithm can realise any *monotone* routing problem, in optimal time, $O(\log p)$ time. A *monotone* routing problem is a *1-relation* for which the relative order of the packets is unchanged. Similarly optimal performance is exhibited by the greedy algorithm for a routing problem for which each message has a destination address which is the complement of its initial address. This *1-relation* can be realised in exactly $\log p$ time on a *p-node* hypercube.

On the other hand, there are bad cases, such as the *bit-reversal permutation* and the *transpose permutation*, for which the greedy algorithm performs very poorly. Consider the *bit-reversal* permutation, routing messages to the address of its initial location read backwards, will take at least $\sqrt{p}$ steps on a *p-node* hypercube. The reason is that there are $2^{\frac{\log p-1}{2}} = \sqrt{\frac{p}{2}}$ nodes with addresses of the form $u_1 u_2 \ldots u_{\frac{(\log p-1)}{2}} 00 \ldots 0$, and all messages starting from such nodes will be at address $00 \ldots 0$ halfway through execution of the algorithm (see section 4.2.1 for a shortest path between any two pairs of nodes in hypercube). The permutations associated with worst-case performance can be overcome by randomising the memory and/or randomised routing.

Any *1-relation* can be converted into a random routing problem by hashing the memory. This is because each message, x, will now be destined for a random location, $h(x)$, where the memory is randomised by the hash function h. Then the greedy algorithm can be shown to perform well on the *butterfly* as in [129, 130]. However, there will still be some permutations which require $\Omega(\sqrt{n})$ time with low probability on the *butterfly*, if we use the greedy algorithm [89].

Valiant proposed a routing algorithm to realise any *1-relation* without altering the memory organisation at all (however, to solve memory contention problems we may need hashing and/or combining). Moreover, the algorithm does not exhibit consistent worst case behavior for any *1-relation* [160, 161, 163]. This algorithm is known as *two phase randomised routing*, and works as follows,

1. For each node i wishing to send a message, choose an intermediate target node

$t(i)$ randomly. That is, for the binary address of $t(i)$ choose each bit to be 0 or 1 with equal probability. Then send the messages by the greedy algorithm,

2. When a message reaches $t(i)$ it is then sent to its true destination, again by the greedy algorithm.

Two phase randomised routing was first shown to work for the *d-hypercube*. With the high probability any *1-relation* can be realised in less than $8d$ (or $8 \log n$) steps [163]. Using this *two phase randomised* routing, subsequent work has been extended to show that, with high probability a *1-relation* can be realised on a *d-cube connected cycles*, *d-butterfly*, *d-shuffle-exchange* and $(d \times d)$ *2-mesh* in $O(d)$ (i.e. proportional to the diameter of the network) steps [7, 88, 138, 154, 158]. Moreover, Valiant has proved the following stronger result for the hypercube.

Theorem 4.3.6 *[158] With high probability, every* $\log p$*-relation can be realised on a p-node hypercube in* $O(\log p)$ *steps.*

An interesting alternative to using *two phase randomised routing* is to use deterministic routing on a randomly wired network. It is shown that deterministically any *1-relation* can be realised on a $\log p$ dimensional *multibutterfly* in $O(\log p)$ steps [90, 155]. Moreover, the routing algorithm on the *multibutterfly* is shown to be robust against faults [90]. Recall that, there is a large number of paths between every input and every output on a *multibutterfly*.

4.3.4 Information dispersal algorithm

The performance of a routing problem can be affected, when a node or a link ceases to transmit data, and/or when a node or a link transmits incorrect data. Moreover, if we use only one path to route a packet then every node and every link must be reliable. An obvious alternative is to create a number of copies of the packet, and route the packets along different paths. This will result in increase in the network load.

Rabin proposed a method in [125]. This is called the Information Dispersal Algorithm (IDA) which breaks each packet into a collection of subpackets such that only a fraction of them suffice to reconstruct the original packet. Since only a fraction of the subpackets have to reach their destination, the algorithm is tolerant to faults. Using the IDA it has been shown that, with high probability, any *1-relation* can be realised on a *p-node* hypercube without any additional delay (i.e. in $O(\log p)$ steps), even if many nodes and links in the hypercube are faulty [61, 97]. Similar results have been obtained for the *de Bruijn* graph [98].

In addition, the IDA can be used to avoid contention in memory [12, 88]. We encode each item in the memory into a number of subitems, k (say), such that the original item can be reconstructed from any fraction of the sub-items, storing each sub-item across k memory modules. Then contention can be avoided by simply dropping subitems, if they occur at hotspots.

4.4 Synchrony and asynchrony

The PRAM model operates in lock step synchrony. There is an implicit synchronisation barrier after every instruction. This lock step synchronisation guarantees that processors reading from global memory can be sure to obtain the correct value from that memory at the end of the previous instruction, easing the task of algorithm design. The PRAM model, however, neglects the cost of this synchronisation. If we add a synchronisation barrier after every step in a feasible parallel model algorithm, then this would increase the complexity by a factor typically of the order of the network diameter. This question is addressed again in chapter 6.

Chapter 5

Embeddings

This chapter describes joint research with A.M. Gibbons and M.S. Paterson. The results described in this chapter on dense edge-disjoint embedding of binary tree were published in [133] and [134]. Similar results for the mesh, which appeared in [51] were merged with those of [133, 134] then extended and published in [135].

5.1 Introduction

Our concern in this chapter is the improvement of running times for PRAM algorithms when implemented on feasible parallel computers in which processing elements with associated memory are located at the node of an interconnection network. The implementation of a large class of PRAM computations on these models can be made to run optimally fast by the employment of certain strategies. For example, if a particular PRAM algorithmic structure is frequently employed, then the idea of embedding this structure in the communication network can lead, as we will show, to optimal implementation on feasible machines. Perhaps the most commonly occurring structure in this regard is the complete binary tree. It is precisely because such logarithmic depth structures are used (either explicitly or implicitly) that polylogarithmic time complexities are often attained for many PRAM algorithms. In this chapter we describe optimally efficient embeddings of the complete binary tree in the following graphs: the hypercube, the de Bruijn, the shuffle-exchange and 2-dimensional mesh.

5.2 Efficiency requirements

In the PRAM model, the complete binary tree is most usually employed as follows. Data for a problem (or sub-problem) are placed at the leaves, and the required result is obtained by performing computations at the internal nodes in one or more sweeps up and down the tree, so that computations at the same depth are performed in parallel. It should be noted however that some algorithms may require simultaneous computation at an arbitrary number of nodes at different depths of the tree. If we are to embed the complete binary tree into the host topology of some distributed memory machine, we therefore need to observe the following requirements to achieve an *efficient* embedding:

1. *All tree nodes at the same depth should be mapped into disjoint interconnection network nodes if (as in the PRAM computation) computations are to be performed in parallel at these nodes. In addition, PRAM algorithms may require computation at nodes of the tree which are of different depth. Thus*

for greatest utility, the embedding should map at most a constant number of tree nodes to any node of the host graph.

2. *Tree edges at the same depth should correspond to edge-disjoint paths in the interconnection network if the commonest types of PRAM algorithm employing this technique are to be simulated. For greatest flexibility, all tree paths should be mapped to disjoint paths in the host graph.*

3. *The maximum distance from the root to a leaf of the tree (in terms of edges of the host graph) in the embedding should be minimised in order that the routing time is minimised.*

4. *Consistent with satisfying the above points, the size of the host graph should be a minimum in the interests of processor economy.*

5.3 The embeddings

In the subsections that follow, we describe embeddings of the complete binary tree with n leaves in the hypercube and de Bruijn graphs and in the *doubly-connected* 2-dimensional mesh and shuffle-exchange graphs, each with n nodes. These are all topologies that have been advocated for interconnection networks and which we individually recall in the following subsections. By doubly-connected, we mean that each edge in the standard definition of the graph is replaced by two parallel edges. As we shall see, with the exception of the shuffle-exchange graph, the embeddings are such as to satisfy the following crucial properties which guarantee that in every respect the efficiency requirements stated in the previous section are met.

Embedding Properties:

1. Each node of the host graph is assigned exactly one leaf of the tree.

2. Each node of the host graph, except one, is also assigned exactly one internal node of the tree.

3. Distinct tree edges are mapped onto edge-disjoint (possibly null) paths in the host graph.

4. Consistent with Embedding Properties 1 and 2, the maximum length of images in the host graph of tree paths from a leaf to the root is optimally short.

In the case of the shuffle-exchange graph, the embedding that we describe ensures that embedding properties 1-3 are satisfied. However, we can only conjecture that embedding property 4 is also satisfied by our embedding. In the embedding, the maximum length of an image of a leaf to root path is $2\log_2 n + 2$, whereas our best lower bound in this case is $(3/2)\log_2 n$.

Let $DRCBT$ be shorthand for *Double Rooted Complete Binary Tree*. A $DRCBT$ is a complete binary tree in which the path (of length 2) connecting the two children of the root is replaced by a path, P, of length 3. Each of the two internal nodes of P (both of degree two) is a root of the $DRCBT$. These roots will be denoted by r_1 and r_2. In the hypercube, de Bruijn and shuffle-exchange graphs, each with n nodes, we shall in fact embed the $DRCBT$ with $2n$ nodes.

The following subsections establish that Embedding Properties 1-3 hold for the

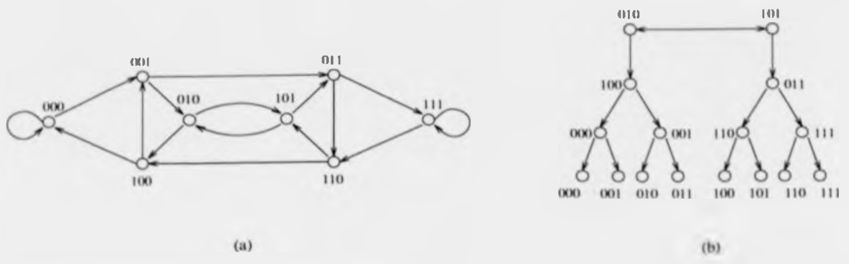


Figure 5.1:

embeddings described. We delay consideration of Embedding Property 4 until the following section. As we shall see, it is also the case that the *multiplicity* of the topologies (that is, the maximum number of parallel edges between any pair of nodes) is a minimum consistent with Embedding Properties 1-2.

5.3.1 Embedding in the de Bruijn graph

The undirected de Bruijn graph of degree m, $m \geq 0$, has $n = 2^m$ nodes which are all the possible distinct binary strings of length m. Each node $b_1b_2\cdots b_m$ is connected to node $b_2b_3\cdots b_mb_1$ by a *shuffle* edge, and to node $b_2b_3\cdots b_m\bar{b}_1$ by a *shuffle-exchange edge*. Here $\bar{b}_k$ is the complement of b_k. By implication, each node is also connected to $b_jb_1b_2\cdots b_{j-1}$ and to $\bar{b}_jb_1b_2\cdots b_{j-1}$).

For our purposes it is convenient to direct the edges of the de Bruijn graph from each node $b_1b_2\cdots b_m$ towards nodes $b_2b_3\cdots b_mb_1$ and $b_2b_3\cdots b_m\bar{b}_1$. This automatically assigns a direction to every edge of the graph and ensures that each node has both out-degree and in-degree of 2. figure 5.1(a) shows the directed de Bruijn graph of degree 4.

It is also convenient here to direct the edges of the $DRCBT$. All edges are directed away from the roots (the edge between the roots will be bi-directed) as the example of figure 5.1(b) illustrates. We say that each directed edge is from a parent to a child.

For i, j both non-negative integers and $i \leq j$, we now inductively define digraphs $G(i, j)$ as follows:

1. $G(0, j)$ consists of two isolated vertices each denoted by a binary string of length j consisting (for positive j) of alternating 0s and 1s, one string begining with a 0 and the other begining with a 1.

2. $G(i, j)$ is constructed from $G(i-1, j)$ as follows. From each vertex $v = b_1 b_2 \cdots b_j$ of $G(i-1, j)$ we add new directed edges (if they do not already exist) to (possibly new) vertices $b_2 b_3 \ldots b_j 1$ and $b_2 b_3 \ldots b_j 0$. The former is called the *left-child* of v and the latter the *right-child* of v.

Lemma 5.3.1 *For $i < j$, $G(i, j)$ is a directed $DRCBT$ and for $i = j$, $G(i, j)$ is a directed de Bruijn graph.*

Proof First suppose that $i < j$ and, to avoid trivial cases, that $j > 2$. Let $[01]_k$ denote either of the binary strings of length k consisting of alternating 0's and 1's that begins with a 0 or a 1. Now, $G(1, j)$ is easily seen to be the directed $DRCBT$ with four nodes. The roots are of the form $[01]_j$ and are connected by anti-parallel edges. The two additional nodes are $c_1 = [01]_{j-2}00$ which is a right-child of one root and $c_2 = [01]_{j-2}11$ which is the left-child of the other. As long as $i < j$ the inductive construction of $G(i, j)$ is such as to grow complete out-trees rooted at c_1 and c_2, each of depth i. To see this, it is sufficient to show that at each inductive step in which $G(i,j)$ is constructed from $G(i-1,j)$, the only new edges connect leaves of $G(i-1,j)$ to nodes whose labels are distinct from all previously obtained nodes and are distinct amongst themselves. Thus, the new nodes will be leaves of $G(i,j)$. It is easy to see that the new nodes are of the form $[01]_{j-1-i}11\alpha$ or $[01]_{j-1-i}00\alpha$, where α is a binary string of length $(i-1)$. These are distinct from all previous labels because, starting at the i^{th} position from the right, they contain either the substring 00 (if they are descendents of c_1) or the substring 11 (if they are descendents of c_2) and all previously existing nodes contain either 10 or 01 at this position. Any two of the new nodes descended from the same c_i will have different αs because each such α is uniquely determined by the path sequence of left-child, right-child moves that must be traced from c_i.

Thus, we have proved that for $i < j$, $G(i,j)$ is a directed $DRCBT$. By a trivial proof, if $i = j-1$ the $DRCBT$ has 2^j distinctly labelled nodes. Because this is the maximum number of distinct binary strings of length j, it follows that $G(j,j)$ will have the same set of nodes as $G(j-1,j)$. Also, every leaf of the $G(j-1,j)$ will have the form 00α or 11α, so that the rightmost substring of length $(j-1)$ is distinct amongst the labels of the leaves. This ensures that in the inductive construction of $G(j,j)$ from $G(j-1,j)$ the new edges (all directed from leaves of $G(j-1,j)$) will be directed to *distinct* nodes. In this way, every node of $G(j,j)$ has in-degree and out-degree of 2. In fact, it trivially follows from the construction of $G(j,j)$ that every node $v = b_2 \dots b_j$ is connected to $b_2b_3 \dots b_j0$ and $b_2b_3 \dots b_j1$ which are the children of v and edges are directed to v from $v_1 = 0b_1b_2 \dots b_{j-1}$ and $v_2 = 1b_1b_2 \dots b_{j-1}$. Of this last pair of nodes, if $b_1 = 0$ then v_1 is a leaf of $G(j-1,j)$ and v_2 is the parent of v in $G(j-1,j)$. If $b_1 = 1$ then the rôles of v_1 and v_2 are reversed. Thus, the

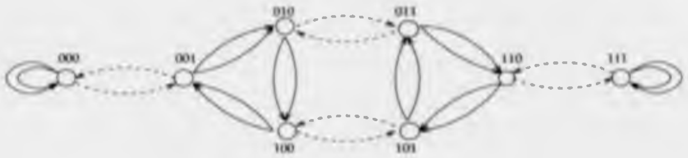


Figure 5.2:

nodal connections of $G(j, j)$ are precisely those of the directed de Bruin graph and this observation completes the proof. □

The following theorem follows trivially from the proof of the preceding lemma.

Theorem 5.3.1 *The directed $DRCBT$ with n leaves can be embedded in the directed n-node de Bruijn graph so as to satisfy Properties 1-3.*

Figure 5.1 provides an illustration of the theorem. Both (a) and (b) are $G(3, 3)$. In (b) each node appears twice, once as a leaf of the directed $DRCBT$ and once as an internal node. Copies of nodes are identified in (a) to show the directed de Bruijn graph.

5.3.2 Embedding in the shuffle-exchange graph

An undirected *shuffle-exchange* graph of dimension m has $n = 2^m$ nodes which are all the possible distinct binary strings of length m. Each node $b_1b_2 \dots b_{m-1}b_m$ is connected by an *exchange* edge to $b_1b_2 \dots b_{m-1}\bar{b}_m$ and by a *shuffle* edge to $b_2b_3 \dots b_mb_1$. By implication, each node $b_1b_2 \dots b_m$ is also connected by a shuffle edge to $b_mb_1b_2 \dots b_{m-1}$.

Figure 5.2 shows the shuffle-exchange graph of degree 6 in which each exchange edges has been replaced by a pair (dashed for emphasis) of anti-parallel edges and each shuffle edge has been replaced by a pair of parallel edges. This particular form is derived from a previously known (see [108], for example) embedding of the de Bruijn graph in the shuffle-exchange graph. The embedding has both *congestion* and *dilation* of 2. The dilation of an embedding is the maximum distance in the host between the images of adjacent guest nodes. The congestion of an embedding is the maximum, over all edges e in the host graph, of the number of edges in the guest graph mapped to a path in the host graph which includes e. The embedding is obtained by removing each shuffle-exchange edge $(b_1 b_2 \ldots b_m, b_2 b_3 \ldots b_m \bar{b}_1)$ from the directed de Bruijn graph and replacing it with the directed path consisting of the shuffle edge $(b_1 b_2 \ldots b_m, b_2 b_3 \ldots b_m b_1)$ followed by the exchange edge $(b_2 b_3 \ldots b_m b_1, b_2 b_3 \ldots b_m \bar{b}_1)$ of the shuffle-exchange graph. Because the graph now employs just the nodal connections of the shuffle-exchange graph, it is precisely such a graph but with parallel and anti-parallel edges. In this way, for example, it is easy to see that figure 5.2 can be derived from figure 5.1(a). The following theorem follows immediately from this embedding of the de Bruijn graph and from theorem 5.3.1.

Theorem 5.3.2 *The $DRCBT$ with n leaves can be embedded in the doubly-connected shuffle-exchange graph with n nodes so as to satisfy conditions* $1 - 3$.

5.3.3 Embedding in the 2-dimensional mesh

The 2-dimensional doubly-connected mesh is the target graph for the embedding of this sub-section. Adjacent nodes are connected by a pair of anti-parallel edges. The guest graph of the embedding is the complete binary in-tree, that is, a complete binary tree in which the edges are directed towards the root. Note that there is a large corpus of work concerning embedding of different classes of graph into the 2-dimensional mesh, for example see [39, 93, 118, 162]. These use different constraints on the embedding form than the ones used here, and address different optimisation issues such as minimising the area of the host graph.

We prove the following theorem [51].

Theorem 5.3.3 *For all $m \geq 1$, there are embeddings of the complete binary tree with 2^{2m} and 2^{2m+1} leaves, into a doubly-connected $2^m \times 2^m$ mesh and a doubly-connected $2^m \times 2^{m+1}$ mesh respectively which satisfy Embedding Properties 1-3.*

Proof First consider the embedding in a square for the tree with 2^{2m} leaves. The case $m = 1$ is easy. For $m = 2$, figure 5.3 shows one possible embedding in the 4×4 mesh. In this figure, the internal tree nodes and the the paths corresponding to the tree edges, are drawn with increasing size and boldness from leaves to root respectively. The edges are directed towards the root. The leaf nodes are not shown explicitly since there is one at each mesh node. Note that some tree edges, incident with the leaves, are mapped to null paths, indicated by loops in the figure. The root is embedded on the left side, but the heavy path shown from this to the top-left corner is used later in larger embeddings. The node distinguished with a dotted square in

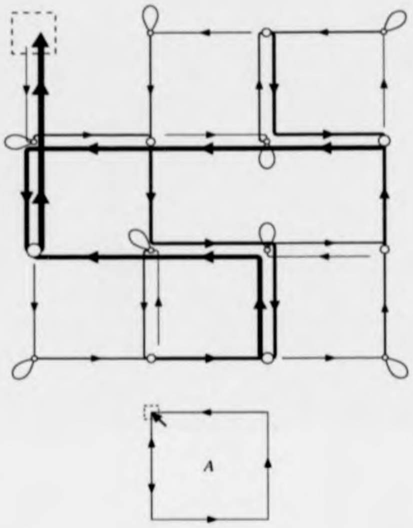


Figure 5.3:

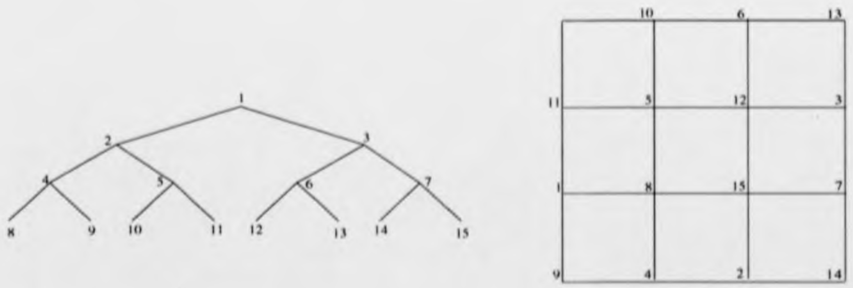


Figure 5.4:

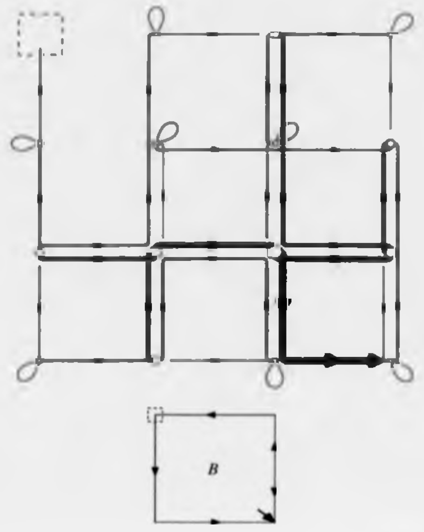


Figure 5.5:

the figure is that unique node which has not yet been assigned an internal tree node. The small diagram underneath gives the salient features of this embedding, A_2, for use in the recursive construction. The arrows on the perimeter indicate the usage so far of the outside edges, and show that all the clockwise outside edges on three of the sides are as yet unused. Perhaps, this embedding can easily understand with the help of figure 5.4. Left hand side of this figure shows the internal nodes of a 32-node (or 2^4 leaves) tree, and the right hand side shows a 4×4 mesh. Each number i, $1 \leq i \leq 16$, of the figure indicates that internal node with number i is mapped into the mesh node with number i.

This construction requires also an alternative 4×4 embedding, B_2, shown in figure 5.5. The root here is embedded in the interior of the square but there is an outgoing path from it to the lower-right corner. This time, all the clockwise edges on the top, left and bottom sides are free.

The next stage in our construction, the embeddings for $m \geq 3$, is shown in figure 5.6. Three A_{m-1}s and one B_{m-1} are combined to give embeddings of the 2^{2m}-leaf tree in a $2^m \times 2^m$ mesh. The three new internal tree nodes required are shown by white and black circles, and are connected by paths of appropriate weight. For the recursion, the embedding is continued in two different ways. the black root node can be connected to the top-left corner by one of the hatched paths shown, or joined to the lower-right corner by another hatched path. The first alternative yields an embedding A_m which has edge characteristics of type A given by the small diagram in figure 5.3, while the second similarly yields B_m. The arrangement shown in figure 5.6 therefore represents a recursive step by which the construction can be continued indefinitely. The third path illustrated, with different hatching, to the lower-left corner, will be used in the 2^{2m+1}-leaf embedding. For the case $2^m \times 2^{m+1}$, if $m \geq 3$ we can connect seven copies of A_{m-1} with one copy of C_{m-1} as shown in figure 5.7. The cases where $m < 3$ are simple. □

5.3.4 Embedding in the hypercube

A hypercube has n nodes (where $n = 2^m$, for some positive integer m) labelled from 0 to $n - 1$ in binary and such that there is an edge between two nodes if and only if their binary labels differ in exactly one bit. For completeness, we briefly present the following result which was first described in [133].

Theorem 5.3.4 *The double-rooted complete binary tree (DRCBT) with $n > 8$ leaves can be embedded in the hypercube with n nodes so as to satisfy Embedding Proper-*

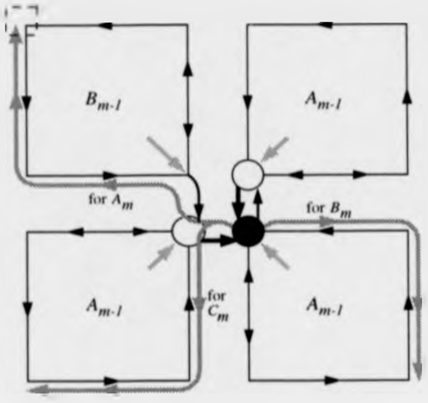


Figure 5.6:

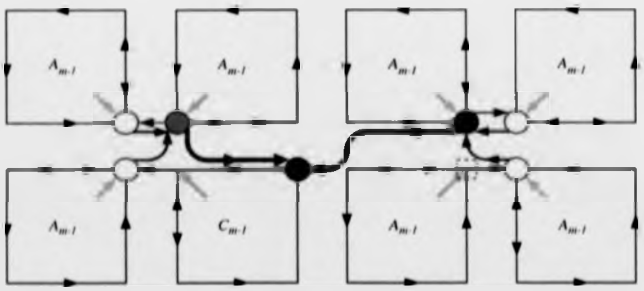


Figure 5.7:

ties 1-3.

Proof For $n > 16$ inductively construct the embedding starting with the base case of $n = 32$, as shown in figure 5.8. In the figure nodes occur at the corners of the squares defined by the dashed lines and the labels of nodes in the top left-hand quarter of the figure are shown. The first two binary bits of the labels of nodes in the other quarters are shown at the center of their quarter of the figure. The last three binary bits of such an address will be the same as the corresponding node in the top left-hand quarter. Generally speaking, figures will only show some edges of the hypercube, just those that are of interest. Dashed edges happen to correspond to certain hypercube edges but are used merely as an aid in locating nodes in the layout. For clarity, two figures (5.8 (a) and (b)) are employed to describe this case. Figure 5.8(a) shows the embedding of those tree edges which have leaves as endpoints. For clarity, some embedded tree edges point towards that endpoint which is a leaf of the tree. Some tree edges are mapped to null paths which are indicated by loops. figure 5.8(b) shows the embedding of all other tree edges. Notice that hashed edges are used for the path of length 3 on which full circles denote the possible roots of the embedded complete binary tree. Also, notice that the three edges on this path belong to three different dimensions of the hypercube. In figure 5.8(b), the internal nodes are drawn with increasing size and the tree edges are drawn with increasing boldness the nearer they are to the root. It is easy to see that this base case satisfies Properties 1-3 in all respects.

Figure 5.9 illustrates the inductive step in the construction of the embedding of the $DRCBT$ with n leaves in the hypercube with n nodes from two embeddings of $n/2$

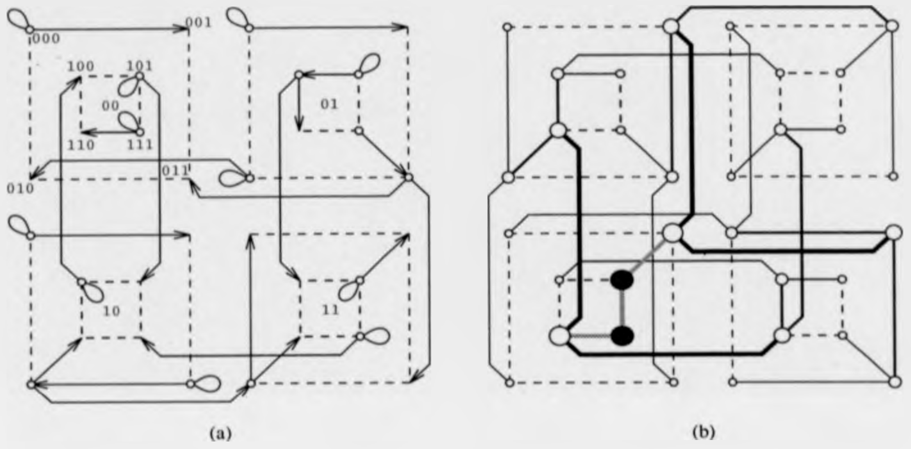


Figure 5.8:

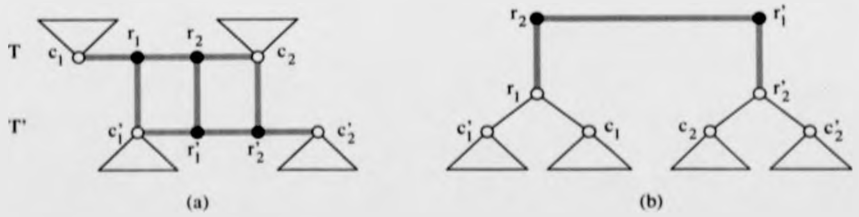


Figure 5.9:

leaf $DRCBT$s in hypercubes with $n/2$ nodes. These two embeddings are denoted by T and T' in figure 5.9(a). The hashed vertical edges in that figure are edges of the new dimension of the constructed hypercube. The hashed horizontal paths $((c_1,r_1,r_2,c_2)$ and $(c'_1, r'_1, r'_2, c'_2))$ are the paths of length 3 which have as internal nodes the possible roots of the embedded complete binary trees with $n/2$ leaves. The triangular shapes attached to children of these possible roots represent the embedded subtrees rooted at these children. The two smaller hypercubes are oriented so that r_1 and c'_1 are made to correspond, then the dimension corresponding with the edge (r_1,r_2) is made to correspond with the dimension of the edge (c'_1,r'_1). In this way, the nodes r_2 and r'_1 are made to correspond. Similarly, the dimension of (r_2,c_2) is made to correspond with the dimension of (r'_1,r'_2) and so node c_2 is brought into correspondence with r'_2. This is always possible given the edge transitivity of the hypercube and given that each of the horizontal hashed paths of length 3 has each edge of different dimension. Figure 5.9(b) shows the embedding of the $DRCBT$ with n leaves and unit dilation in the constructed hypercube with n nodes. The labelling of nodes in this figure makes clear its derivation from figure 5.9(a).

For $n = 16$, an embedding satisfying the Properties 1-3 is shown in figure 5.10. Note that figure 5.10 ((a) and (b)) is the hypercube layout corresponding to the top half of figure 5.8 ((a) or (b)). Again, for convenience of illustration the embedding of tree edges attached to leaves in one diagram (figure 5.10(a)) and the embeddings of all are shown other edges in another (figure 5.10(b)). □

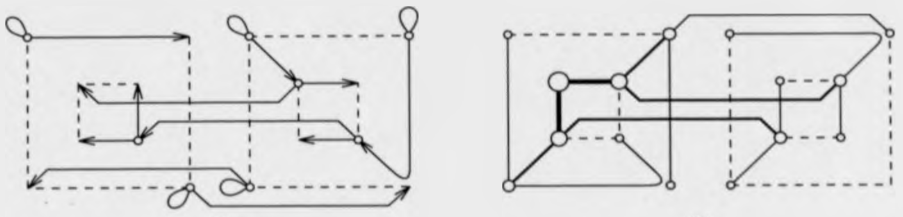


Figure 5.10:

5.4 Depths of the embedded trees

In this section we examine the quality of our embeddings from the point of view of Embedding Property 4. The maximum distance from the root to a leaf in the image of the complete binary tree for any host graph is an important algorithmic parameter. It is a measure of the routing time required for a single sweep of the balanced binary tree. We denote this distance by $P(n)$ for the complete binary tree with n nodes. If the embedding satisfies Embedding Properties 1-3 of Section 5.2, the O(log n) routing time of the PRAM algorithm for such a sweep translates to $O(P(n))$ for the interconnection network.

We first determine $P(n)$ for the embeddings in each of the four interconnection networks considered in this chapter. Then we establish lower bounds for maximum root-to-leaf distances for these embeddings which we use to show that, in all cases except for that of the shuffle-exchange graph, our values of $P(n)$ are asymptotically as short as possible. We conjecture that this fact is also true for the shuffle-exchange graph, although there is a gap between the $P(n)$ of our embedding and the lower bound obtained.

It will also be evident from this section that the congestion and the load of our host graphs are a minimum consistent with satisfying Embedding Properties 1-3. The load of an embedding is the maximum number of the guest nodes mapped to a node of the host graph.

5.4.1 Maximum root-to-leaf distances of the embeddings

For the n-node hypercube and de Bruijn graphs $P(n)$ is $\log_2 n + 1$. This is because each edge of the n-leaf $DRCBT$ is mapped into at most one edge of the n-node host and there are root-to-leaf paths for which every such edge is mapped to precisely one edge of the host. Thus, for the embedded $DRCBT$ the maximum length of root-to-leaf paths is $\log_2 n$ in these cases. This translates to $\log_2 n + 1$ for the complete binary tree when its root is identified with a particular one of the two roots of the $DRCBT$.

In our embedding of the n-leaf $DRCBT$ in the doubly-connected n-node shuffle-exchange graph one of the pair of edges from each parent to its children is mapped into two edges of the host and so for this case $P(n) = 2(\log_2 n + 1) = 2\log_2 n + 2$.

Now consider the embedding of the $n = 2^{2m}$-leaf complete binary tree into a doubly-connected $2^m \times 2^m$ mesh. Let $D(m)$ be $P(n)$ when expressed as a function of m. It is a trivial matter to construct an embedding for $m = 1$ for $D(m)$ is 2. For $m = 2$, we see by inspection of the embedding B_2 of figure 5.5 that this maximum distance is 6 mesh steps and so $D(2) = 6$. Now consider square meshes with $n = 2^{2m}$ nodes, $m \geq 2$. Let A(m), B(m) and C(m), be the maximum distances from a leaf to the output from the top left corner of pattern A_m, the lower-right corner of pattern B_m

and the lower-left corner of pattern C_m, respectively. From figures 5.3 and 5.5, we see that A(2) = 10 and B(2) = 8. A corresponding layout C_2 with C(2) = 9 is easy to derive from B_2. We can verify from figure 5.3 the following recurrence equations for $m \geq 3$:

$$\begin{aligned}
A(m) &= D(m) + 2^m, \\
B(m) &= D(m) + 2^m - 2, \\
C(m) &= D(m) + 2^m - 1, \\
D(m) &= max\{A(m-1) + 2, B(m-1) + 2\} \\
&= D(m-1) + 2^{m-1} + 2.
\end{aligned}$$

The solution to these equations is:

$$D(m) = 2^m + 2m - 2, for\ m \geq 1,$$

that is,

$$P(n) = \sqrt{n} + 2\log_2 n - 2 = \sqrt{n} + O(\log n)$$

Now consider the case of embedding the $n = 2^{2m+1}$-leaf tree into a doubly-connected $2^m \times 2^{m+1}$ mesh. Let $D'(m)$ be the corresponding maximal leaf-to-root distance. We may verify in figure 5.7 that:

$$D'(m) = max\{A(m-1) + 4, C(m-1) + 3\} + 2^{m-1}$$

and so in this case,

$$P(n) = \frac{3}{2}\sqrt{\frac{n}{2}} + O(\log n)$$

5.4.2 Lower bounds for the embedded tree depths

Here we obtain lower bounds for the depth of the complete binary tree for the different embeddings of this chapter and show that, in the cases of the mesh, hypercube and de Bruijn graphs, these bounds asymptotically match the values of $P(n)$ that were obtained in the previous subsection.

For a given graph, let its *radius*, ρ, be the minimum distance r such that for some *central* node c, every node is at a distance at most r from c. Clearly, for any embedding of a complete binary tree in a communication network in which Embedding Properties 1-3 are met, a lower bound for $P(n)$ is provided by ρ.

Lemma 5.4.1 *The following relationships hold for graphs with n nodes: for the hypercube $\rho = \log_2 n$ and for the shuffle-exchange graph $\rho = \frac{3}{2}(\log_2 n - 1)$ for $\log_2 n$ odd and $\rho = \frac{3}{2}\log_2 n + 1$ for $\log_2 n$ even.*

Proof Follows easily from the definitions. □

For the hypercube we can obtain a marginally stronger lower bound for $P(n)$ from the following Lemma.

Lemma 5.4.2 *Consider leaf-disjoint embeddings of a complete binary tree with n leaves into the n-node hypercube. For any such embedding $P(n) \geq \log_2 n + 1$.*

Proof If there were an embedding satisfying Embedding Properties 1-3 and $P(n)$ was $\log_2 n$, then this would imply that a unit dilation embedding of the complete binary tree (perhaps with some leaf to parent edges mapped to null paths) was possible in the hypercube. Consider the mapping of the subtree consisting of all edges other than leaf to parent edges. An embedding of this subtree would have to be vertex disjoint with every edge being mapped to a hypercube edge, this is not possible because such a graph is not a subgraph of the hypercube as follows. Both the subtree and the hypercube are bipartite graphs. In the case of the hypercube both halves of the bipartition contain the same number of nodes, this is not the case for the subtree and (with each subtree edge mapped precisely to an edge of the hypercube) this would force more than one node of the subtree to be embedded in a single node of the host. □

Lemma 5.4.3 *Consider leaf-disjoint embeddings of a tree with n leaves into the mesh. For an arbitrary embedding*

$$P(n) \geq \sqrt{\frac{n}{2}} - O(1),$$

for an embedding into an $r \times s$ rectangle where $n = rs$,

$$P(n) \geq \lceil \frac{r-1}{2} \rceil + \lceil \frac{s-1}{2} \rceil = (r+s)/2 - O(1).$$

Proof Let b_r be the number of vertices of the mesh graph $\mathcal{Z} \times \mathcal{Z}$ within path length r of the origin, where $\mathcal{Z}$ is the set of integers. Then $b_0 = 1$, $b_1 = 5$, and in general $b_r = 1 + \sum_{i=0}^{r} 4i = 2r(r+1) + 1$ for $r > 0$. For any injective mapping of n leaves into the mesh, if $n > b_r$ then some vertex has to be mapped to a mesh node at distance greater than r from the root. □

From the above Lemmas, and consistent with Embedding Properties 1-2, it follows that the values of $P(n)$ attained by our embeddings are asymptotically as short as possible for the de Bruijn, two-dimensional mesh and hypercube interconnection networks. Thus for these graphs, Embedding Property 4 has been optimally met. We conjecture that Embedding Property 4 has been optimally met also for the shuffle-exchange graph, although in this case the $P(n)$ of our embedding is asymptotically a constant factor of $\frac{4}{3}$ greater than our lower bound.

5.5 Further remarks and algorithmic issues

Here we briefly justify the use of parallel or anti-parallel edges in our embeddings, where they occur, so long as the embeddings have the density implied by Embedding Properties 1-2. We then comment on the complexity gains afforded by our embeddings when they might be employed for PRAM implementation on the associated communication networks.

A natural question to consider is whether the pairs of anti-parallel edges are necessary for the mesh. Can the complete (undirected) tree be densely embedded in the usual undirected mesh? Each mesh node (except two) is host to one leaf vertex, with degree one, and one internal vertex, with degree three, and so has a total of at least four embedded edges incident with it. Note that some of the edges adjacent to leaves can be mapped into paths of length zero, the loops in our figures, and so some mesh nodes may require only *two* of their incident mesh edges. Thus there is no immediate contradiction from degree considerations. However, we now consider local details and easily find a contradiction. Consider boundary mesh nodes, away from the one special node that does not host an internal tree vertex. Any such node has degree less than four and so must have a loop in the embedding. It therefore is host to a leaf vertex and the internal node adjacent to that leaf, and requires one incoming path from another leaf, and one outgoing path to the parent vertex. Since the neighbouring boundary nodes are in the same predicament, there is an impossible situation at the boundary, even worse if it is at a corner.

It is also easy to see that we need parallel (or anti-parallel) edges for the shuffle exchange graph when embedding the complete binary tree if the embedding is consistent with Embedding Properties 1-2. This is because the shuffle exchange has degree 3 but any internal node of the tree which is not adjacent to a leaf has to be mapped to the same node of the shuffle exchange graph as a leaf. This requires that at least four tree edges have this shuffle exchange node as an end-point which is not possible without parallel (or anti-parallel) edges being added to the shuffle exchange graph to ensure edge-disjointness of the embedding. Notice that it also then follows that the dilation of the embedding must be greater than 1.

For the hypercube, de Bruijn and shuffle exchange graphs our embeddings show that the complete binary tree can be edge-disjointly embedded in hosts that are generally half the size compared with previously described embeddings without detriment to the time complexities of PRAM algorithmic implementations that employ the complete binary tree. The embedding of a complete binary tree in the hypercube described in [16] meets all our efficiency requirements except that the host graph is twice as large as it need be. In fact [16] embeds the n leaf complete binary tree in the hypercube with $2n$ nodes. In [88] (pages 407-410), an embedding is described in which the n leaf tree is embedded in the n node hypercube. However, in this embedding, up to $\log_2 n$ tree nodes of different depths are mapped to a single node of the hypercube. Although the embedding is such as to facilitate the efficient implementation of most PRAM algorithms, there may be difficulties in the exceptional cases when simultaneous computation is required to take place at an arbitrary number of different levels within the tree. Such an example is cited later in this section.

For the two-dimensional mesh, our embeddings may not only reduce the size of the host graph but will also improve running times of the implementations. For example, in the well-known H-tree construction (see for example, page 84 of [153]), the complete binary tree with n leaves is embedded in the $(2\sqrt{n} - 1) \times (2\sqrt{n} - 1)$ mesh and the maximum root to leaf distance in the mesh image is $2\sqrt{n} - 2$. Of course, this embedding was not designed from the point of view of our criteria and would in any case be very costly in terms of unemployed processor sites. In the embedding of [54], although the complete binary tree with n leaves is embedded in the square mesh with n nodes, the maximum root to leaf distance is $3.54\sqrt{n}$. Moreover, only tree edges at the same depth are mapped to disjoint paths.

Compared with previous embeddings and for some PRAM algorithms, the edge-disjointness property of our embedding in the mesh yields further complexity gains. Occasionally it is useful for all nodes in the tree, not just those at the same level, to pass messages simultaneously to their children in such a way that this continues until all messages (including that from the root) reach the leaves of the tree. An example of such a *cascading* requirement is provided within the implementation of a bracket matching algorithm on a mesh detailed in [54]. This can be simulated in the embedding of [54] by allowing the messages from the internal tree nodes adjacent to leaves to be passed directly to the leaves, then subsequently messages from the nodes at the next level are sent to the leaves and so on, until finally the message from the root is allowed to be copied down to all descendants. In this way, only tree edges at the same level are being employed at the same time and the lack of disjointness of all paths from the root to the leaves is no hindrance. In the embedding of [54], the routing time for such a process would be $3.54(1 + 1/2 + 1/4 + 1/8 + \cdots + 1/2^{\log n})\sqrt{n} \approx 7.08\sqrt{n}$. The successive terms in the series arise from routing from successive tree levels. This is because in successive iteration the size of each subproblem and the required area of the mesh for each subproblem are recursively reduced by a factor of 2. For the embedding of this chapter, the path-disjointness property allows messages to be passed down the tree simultaneously from all levels, and so the routing time for the *cascading* requirement is just that for passing a message from the root to the leaves (this masks the time for message passing from all other internal nodes) which is $\sqrt{n}$.

5.6 Summary and open problems

We have described dense edge-disjoint embeddings of the complete binary tree with n leaves in the following n node intercommunication networks: the hypercube, the de Bruijn and shuffle-exchange graphs and the 2-dimensional mesh. The embeddings have the following properties: paths of the tree are mapped onto edge-disjoint paths of the host graphs, at most two tree nodes (just one of which is a leaf) are mapped onto each host node. We also proved (except for the shuffle-exchange graph) that an algorithmically important parameter, the maximum distance from a leaf to the root of the tree, is asymptotically as short as possible. We conjecture that for the shuffle-exchange graph this distance is also optimally short within our embedding. The embeddings facilitate efficient implementation of many PRAM algorithms on these networks and improve extant results. For the mesh and shuffle-exchange graphs these embeddings were not possible without replacing each edge by a pair of parallel (or anti-parallel) edges.

A number of problems remain open. Because of the logarithmic lower order term in $P(n)$ for the embedding of a complete binary tree in the mesh there is a small gap between the distance obtained here and the naive lower bound of the network radius. Whether this gap can be closed, from either side, is an open question. A mesh architecture sometimes used is in the form of torus, with no boundary. It seems unlikely that a complete binary tree with 2^{2m} leaves could be embedded in the directed $2^m \times 2^m$ torus, but we have not been able to prove this. There is also no proof of our conjecture that for the shuffle-exchange graph our embedding exhibits a shortest possible maximum root to leaf distance consistent with our other embedding requirements.

The question of how to find similarly dense embeddings of complete binary trees in meshes of higher dimension is unanswered. Similarly, the question of finding dense embedding of complete trees of fixed higher degree in communication networks is insolved. From a graph-theoretic point of view, dense edge-disjoint embeddings of *arbitrary* trees in communication networks present a challenge, although these problems may prove to be of less general algorithmic importance than embedding complete trees.

Chapter 6

Practical Parallel Models of Parallel Computation

The work described in this chapter was carried out as part of the ESPRIT I project PUMA which was supported by the European Community (under project number P2701) and appeared in a report [132] to the European Commission. This chapter reviews the Models of Massively Parallel Computation that have been proposed to investigate the major issues in practical parallel computation (including scalability, granularity, asynchrony, latency, fault tolerance, contention and congestion which were introduced in chapter 4). None of these models has yet achieved a consensus as a target model for both hardware and software design, but each provides its own lessons for the design of efficient, fast, reliable parallel software. With the use of theoretical solutions which are described in chapter 4, this chapter demonstrates that there is no theoretical hindrance in designing massively parallel models of parallel computation.

6.1 Introduction

In the previous chapter we described architecture dependent embedding techniques which can lead to optimal PRAM algorithmic PRAM simulation on feasible machines. Such techniques, although of wide usefulness, are not always applicable. In this chapter we address the important question of bridging models for general purpose parallel computing which (like the von Neumann machine in sequential computation) can act as an interface between the actual hardware and the PRAM model. Thus software issues will be separated from hardware issues and the prospect of genuinely portable software in an environment of user friendly high level coding would be a possibility. In other words, from the programmer's point of view, the realistic parallel model can be made to appear like a PRAM.

There have been several models proposed to bridge the gap between the PRAM model and feasible machines. These models variously take account of communication latency, contention and congestion, asynchrony and/or component failures. They have been introduced together with simulation results, demonstrating the extent to which PRAM algorithms can be implemented with little or no asymptotic loss in efficiency. In particular, a user can view these models as extended PRAMs, which hide hardware details from the user. We shall briefly describe some of these models and survey some of the difficulties involved in simulation. By doing so, we show the possibility of general purpose parallel computing.

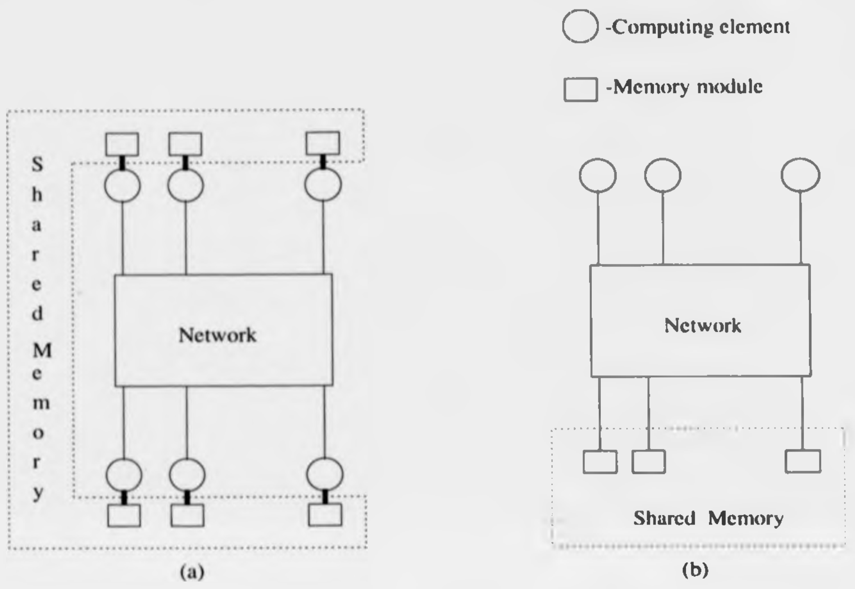


Figure 6.1:

6.2 The practical PRAM model

The practical PRAM consists of a set of components or processing elements connected by a network. Each processing element consists of a computing element and a memory module, as in figure 6.1(a). We may even consider that, the computing elements and the memory modules are separated as in figure 6.2(b). In both models, all the memory modules can be considered to be *global memory* or *virtual shared memory*. Each computing element can access any non-local memory module through the network.

From an algorithm designer's view, the network becomes a "*black box*". Reads and writes to and from the non-local memory module by a computing element are subject to uniform delay, or *communication latency*, which is a parameter l of the machine. As a function of p (the number of processors), l can vary depending on the architecture (e.g. it might be $O(\sqrt{p})$ for the $\sqrt{p} \times \sqrt{p}$-node 2 dimensional mesh, or $O(\log p)$ for a p-node hypercube). By uniform delay, we mean that the access time to a non-local memory module is independent of the processor making the request.

There is a *router* for routing data between computational elements and memory modules. As computing elements and (non local) memory modules communicate through a router [157], the tasks of computation and communication can be separated. The router operates independently of the individual processors. Once a data packet is delivered to the router (through a router interface), the packet is routed through the network to its destination without any burden on the processing elements which may continue their processing. Note that, by separating computation from communication , no particular network topology is favoured beyond the requirement that a high throughput be delivered.

For example, if computational elements want to access distinct non-local memory modules, realising a *1-relation*, then they send the requests to the router through a router interface. As we saw in section 4.3.3, a *1-relation* can be realised in $O(l)$ time, where l is the diameter of the network. Note that the algorithm designer does not need to know about the topology of the network.

The global memory of the feasible machine is divided into a number of memory modules, in contrast to the PRAM's memory which is a single block. Moreover, accessing a global memory location will take the $O(l)$ steps on the practical PRAM, whereas the time is $O(1)$ on the PRAM. However, the practical PRAM and PRAM are similar to the user, since both models have no notion of network locality. In this chapter we survey *optimal* simulations of PRAM algorithms on the practical PRAM, and argue that the practical PRAM could be a candidate for general purpose parallel computing. By *optimal* simulation we mean that when simulating a PRAM algorithm on a practical PRAM the work done for the algorithm on the PRAM is equal to, within a constant factor, the work done on the practical PRAM.

6.3 Latency hiding

A naive approach to implementing a PRAM algorithm on the practical PRAM is to allow $O(l)$ time for message routing after every step of the PRAM algorithm. This significantly slows down the algorithm. Techniques are needed to "tolerate" or "hide" the network latency. Recently, a series of results in [1, 2, 24, 55, 83, 116, 117, 157, 158] have shown how parallel algorithms for realistic models can be designed such that the effect of network latency can be minimised with respect to work measure.

First we consider [2, 116, 117], which capture the communication and computational complexity of PRAM algorithms. The model used is called the *Local-memory Parallel Random Access Machine* (LPRAM) [2]. The LPRAM is a CREW PRAM in which as well as global memory each processor is provided with an unlimited amount of local memory. As in our practical PRAM, the LPRAM has a parameter l which is the time taken by one communication step. Each computation step time takes unit time . The total time of an algorithm is $T + (l \times C)$, where T is the number of computations steps and C is the number of communication steps. The different model of [117] can be thought of as a pipelined version of the LPRAM .

The computational problem to be solved is presented as a data-dependency graph. The data dependency graph is a directed acyclic graph (DAG). We model the computational problem to be solved as a DAG, with its nodes corresponding to operations and its arcs corresponding to the values computed by performing such operations. A *computation schedule* of the DAG consists of a sequence of computation steps and communication steps. At a computation step each processor may evaluate a node of the DAG; this evaluation can only take place when its local memory has the values of all incoming arcs into this node. At a communication step, any processor may write into the global memory any value that is presently in its local memory, and then it may read into its local memory a value from global memory.

A node of the DAG with in-degree zero corresponds to the value of an input. The inputs are initially stored in the global memory, and the output of the DAG has to be written into the global memory. Our problem is to efficiently schedule a DAG to minimise overall computation time and communication time such that the total time is minimum. For example figure 6.2 shows a DAG and the schedule that computes the DAG with two processors, P and Q.

This schedule computes the DAG in five communication steps (i.e. $C = 5$) and three computation steps ($T = 3$), the total computation time is $3 + 5l$. Note that if we allow Q, which is idle at the communication steps 1 and 2, to read a and b, then communication step 3 is not necessary. Hence, the total time is reduced to $3 + 4l$. In other words, allowing several processors to compute the same value can save some communication at no additional time delay. It is an open problem whether recomputation can save more than a constant factor in time or communication.

Comm. step1 : P reads a

Comm. step2: P reads b

Comp. step1: P computes c

Comm. step3: P writes c, Q reads it

Comp. step2: P computes d; Q computes e

Comm. step4: Q writes e, P reads it

Comp. step 3: P computes f

Comm. step5: P writes f

$$\begin{tikzcd}
a \arrow[dr] & & b \arrow[dl] \\
& c \arrow[dl] \arrow[dr] & \\
d \arrow[dr] & & e \arrow[dl] \\
& f &
\end{tikzcd}$$

Figure 6.2:

Moreover, it is an NP-complete problem to decide, given a DAG, an integer l, and T_{max}, whether there exists a schedule S such that no time greater than T_{max} is used [117].

In [116] nontrivial trade-offs between communication and computation were shown for the diamond DAG. Those results were not satisfactory because no general principle or technique was described which is applicable to all DAGs. But, in [117] the technique was generalised to all DAGs and the technique was applied to three particular families of DAGs: the complete binary tree, butterfly, and the diamond. Aggarwal *et al.* [2] have shown that matrix multiplication can be modelled as a DAG in the form of a complete binary tree. Furthermore, they obtained (upper and lower) bounds for any binary tree DAG (in which each internal node has exactly two children). This DAG is determined by the design of the algorithm but none of [116, 117, 2] have described a general technique for algorithmic design for these types of trade-offs. However, they show that, if we schedule a DAG such that temporal locality of reference is utilised then the communication cost can be reduced.

Aggarwal, Chandra and Snir introduced a model called the *Block PRAM* (BPRAM) in [1]. They show that efficiency can be enhanced by using temporal and *spatial* locality of reference. *Spatial* locality of reference is that data items to be accessed by a processor are in contiguous locations in the global memory. The BPRAM is similar to our practical PRAM and LPRAM, in which a processor may access a block of contiguous locations from global memory, and it may write a block into contiguous locations of the global memory. Recall that access to a global memory location may take up to $O(l)$ units time. On the other hand, a block of b consecutive words in global memory (or local memory of another processor for the model of [83]) can be copied into local memory, optimally, in time $l + b$ and vice versa. For example, l consecutive words in global memory can be accessed in $O(l)$ time, not in $O(l^2)$ time. To support this, recent randomised routing algorithms provide strong theoretical support [4, 91]. The router can realise any permutation of p messages (i.e. a *1-relation*) of size m in time $O(m + \log p)$ with high probability on a *p-node* hypercube. This is because, there is a significant overhead for establishing a communication. Once established, a large amount of information can be transferred at low cost. The latency can be hidden in this way by *pipelining* a block of global memory access.

We can implement block pipelining to hide latency provided that each processor has multiple requests to global memory at the same time, and requests can be grouped into blocks of length $\Omega(l)$. This can be resolved by assigning many PRAM processors ("virtual processors") to each actual processor ("physical processor") of the machine. The ratio v/p is called the *parallel slackness*, here v is the number of "virtual processors" and p is the number of "physical processors". Instead of executing one process on each processor, we now provide each processing element with a scheduler allowing it to share its time between v/p processes. In other words, the algorithm is written for v virtual processors, where v significantly exceeds p, the number of physical processors. Then each physical processor may make many requests during each simulation step. Note that here the requests are made to blocks of contiguous locations in global memory

Many PRAM algorithms can be restructured specifically to provide for block accesses using $O(l)$ parallel slackness (i.e. $v \geq lp$) [1, 24]. For example, consider the problem of transposing a $\sqrt{v} \times \sqrt{v}$ matrix. The matrix is given in *row major order* $a_{1,1}, a_{1,2}, \ldots, a_{1,\sqrt{v}}, a_{2,1}, \ldots, a_{\sqrt{v},\sqrt{v}}$ in the first v locations of global memory, and the output $a_{1,1}, a_{2,1}, \ldots, a_{\sqrt{n},1}, a_{1,2}, \ldots, a_{\sqrt{n},\sqrt{n}}$ is desired in the next v locations. This computation can be performed on an EREW PRAM in $O(1)$ time using v processors.

On the BPRAM the transposing computation can be done in $O(v/p + l\sqrt{v/p})$ time using $p \leq v$ processors. During the computation each processor is assigned to transpose a submatrix of size $\sqrt{v/p} \times \sqrt{v/p}$. That is each physical processor is doing the "job" of v/p virtual processors. The algorithm executes in *rounds*, which are either *read rounds* or *write rounds*. Each row of a submatrix requires a separate block read operation from global memory, likewise each column requires a separate block write operation. During a read round each processor reads a block of size

$\sqrt{v/p}$ (i.e. a row of the assigned submatrix) from global memory, taking $l + \sqrt{v/p}$ time. Since each submatrix has $\sqrt{v/p}$ rows there are $\sqrt{v/p}$ read rounds. Thus all the read rounds take $O(v/p + l\sqrt{v/p})$ time, similarly all the write rounds take the same time. Note that each block access is to consecutive locations of global memory.

This BPRAM algorithm is an optimal algorithm if $v/p > l^2$. In this case the work done by the PRAM, $W_{PRAM} = O(1) \times v$, is equal, to within a constant factor, to the work done by the BPRAM, W_{BPRAM}, where $W_{BPRAM} = O(v/p + l\sqrt{v/p}) \times p = O(v)$.

In general, a BPRAM takes up to l times as long as to run its EREW PRAM counterpart. However, the factor of l can be reduced in the BPRAM for several problems. These include matrix transposition, matrix multiplication and Fast Fourier Transform [1]. The factor of l occurs for problems (for example, performing general permutations on elements in memory [1]) that have *fine granularity* [84]: the computation can not efficiently use large blocks for communication. This can either be due to poor spatial locality: data items to be accessed by a processor are not in contiguous locations, or due to poor temporal locality: successive accesses can not be blocked together, e.g. because of control dependencies. In particular, Chin justifies the claim of Gazit, Miller and Teng [47] that list ranking procedure should be replaced with prefix sum whenever possible on the BPRAM [24]. This is because prefix sum has better locality of reference than list ranking.

Now consider ***arbitrary pipelining***, instead of block access to contiguous locations in global memory, a processor can access **any** h locations distributed in global memory. Block pipelining has the practical advantages that a batch of values are guaranteed to return in order, and the entire line or cache line or memory page can be transferred to local memory as a unit. However, arbitrary pipelining is clearly more flexible than block pipelining. It is also easier to use, since the programmer does not have to ensure that values of interest are collected into contiguous locations. That is, the programmer does not need to worry about locality of references.

The *Bulk Synchronous Parallel* (BSP) model of Valiant [157] and the *Phase PRAM* of Gibbons [55] allow arbitrary pipelining. Computation on these models proceeds in a sequence of *supersteps*. In each superstep each processing element is given a task that can be executed using the data that is already available locally before the start of the superstep. The task can be computation, message transmission or message receipt. Let L denotes the time to complete a superstep.

Assume that a router can realise any *h-relation* in gh time, where g is the throughput of the router. Then we choose L to be at least gh. That is, in a superstep L local operations, or an *L/g-relation* can be realised. Note that an *L/g -relation* can be realised in L time because $L \geq gh$.

To support the BSP model Valiant provides a strong theoretical result. Theorem 4.3.6 of chapter 4 shows that a $\log p$ relation can be realised on a *p-node* hypercube in $O(g \log p)$ time. This gives an optimal simulation. A superstep of the BSP model can be simulated in $O(L)$ time, when the topology of network is a hypercube every node of which is a computing element with memory, and $L \geq g \log p$. In each superstep L local computations are performed or an *L/g-relation* is realised. All this can be done in $O(L)$ time on the hypercube.

6.4 Asynchronous computation

It is clear that a PRAM algorithm can be converted into an asynchronous PRAM algorithm by imposing synchronisation after each statement of the algorithm. However, this significantly slows the algorithm down. In this section we investigate the design of algorithms which minimise the cost of synchronisation.

Recent papers [33, 101, 113] focus on the *implicit cost* of synchronisation. For example, in [101] the time complexity of an asynchronous algorithm is the number of instructions executed by a processor including busy wait instructions (i.e. taking the implicit costs of synchronisation into account). The model used is the CRCW PRAM, but the processors can have arbitrary asynchronous behavior, including arbitrary unbounded delays in executing instructions. These delays can be overcome through the use of randomisation as follows. In this model processors are not assumed to have unique IDs; each processor is instead equipped with an independent random number generator. A directed acyclic graph (DAG) representing the tasks to be performed, and the dependencies between them are placed in the shared memory. Each processor selects a task at random, performs the task if its predecessors in the graph have been completed, and repeats. In this way, processors that are delayed, or that have failed, do not unduly slow down the computation: the fast processors will simply evaluate more nodes in the graph. For example, a maximum finding algorithm proceeds in the following manner:

1. *Examine the root node. If it has been evaluated, exit the computation*
2. *Select an interior node uniformly at random*

3. *If the children of the interior node have been evaluated, evaluate the node*

4. *Return to step 1*

By this method any n-processor PRAM algorithm that solves a DAG in $O(T(n))$ time can be transformed into an asynchronous computation with $O(nT(n))$ expected work using $n/\log\log^* n$ processors [101]. Hence the simulation is optimal. However, the model does not account for communication delay. If there is a communication delay (as in practical machines) then choosing a node of a DAG randomly will not minimise the communication delay.

The APRAM, introduced by Cole and Zajicek [32] focuses on the *explicit costs* of synchronisation. The goal of the APRAM model is to design algorithms which avoid global synchronisation, thereby reducing the explicit costs of synchronisation. In this scheme one time unit is called a *round*, which was introduced earlier in [96, 10]. In one round each processor executes at least one instruction, the slowest executes one and faster processors execute more. For instance in the parallel summation problem, the algorithm uses $2n - 1$ (shared) memory location treated as an implicit complete binary tree. Each location is assumed to contain an extra *valid* bit; the valid bit of the input value is assumed to be initially true and all other locations start with a valid bit of false. The algorithm terminates when the valid bit of the root is true. The algorithm for process i is given below, where $L(i)$ and $R(i)$ are respectively the left and right children of node i:

1. *wait until (L(i) is valid)*

```
2. wait until (R(i) is valid)

3. V(i) := L(i) + R(i)

4. valid(i) := true
```

The sum of n numbers can be computed in $O(\log n)$ rounds, the worst case is achieved when every processor executes a single step each round. The APRAM permits multiple sets of processors to synchronise independently and in parallel, it does not account for communication delay, and it permits concurrent reads and writes. This measure does not capture the extent to which slowing down a subset of the processors slows down the overall running time of the algorithm. They introduce a complexity measure by dividing processes into two sets, the slow and the normal processes. For example, the summation algorithm takes $O(\log n) + f(s, c)$, where f is a function of s and c. Each of the slow processes executes at least one event in any s consecutive rounds, where c is the number of slow processes. However, this measure is difficult to analyse and to implement on practical machines.

Many asynchronous algorithms have been developed for particular problems [32, 33, 101, 113, 126, 57]. Most of this work is tailored to specific machines and does not present a general treatment of asynchronous parallel computation. Moreover, communication delays have not been considered for developing asynchronous algorithms. As in feasible models, delay for communication will not give reasonable time complexity for their algorithmic techniques.

Gibbons [55] suggested an asynchronous PRAM model, the Phase PRAM, which includes extra hardware needed to achieve synchronisation. The set of operations between two synchronisation barriers is called a *phase*. The cost of a phase is the maximum number of steps taken by any of the processors during that phase, and the cost of a synchronisation barrier is B(p), which is a function of the number of processors.

Valiant's BSP model [157] incorporates barrier synchronisation in a similar way to the Phase PRAM. Each processor operates in accordance with a barrier synchronisation protocol which may be supervised by a master synchroniser. The synchroniser ensures that all (or subset of) the processors (and the router) have completed a superstep (that is a phase) and, if so, signals all processors to continue to the next superstep.

In essence, the insertion of synchronisation barriers between supersteps ensures that the algorithm is slowed down to the speed of the slowest processor within each superstep. Although the number of steps executed remains the same regardless of the relative speeds of the processors, the time elapsed to execute this type of algorithm is in fact very sensitive to changes in speeds of processors. In particular, if the slowest processor is very slow, then so is the actual running time of the algorithm. It is a weakness that such considerations are not reflected in the complexity measure.

However we can overcome the problem by synchronising all or a subset of the components at regular intervals of L time units where L is a periodicity parameter. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the processors. If it has not, then the next period of L units is allocated to the unfinished superstep. The results of the runtime analysis will not change by more than small constant factors.

Synchronisation is needed in a PRAM algorithm precisely when a write to a shared memory location is followed by a read to the same location. Provided there is no need for synchronisation, processors running asynchronously can execute L instructions each between synchronisation barriers. In this way processors in an asynchronous machine can perform L instructions, followed by a synchronisation barrier cost $\leq L$, not in time $O(L^2)$, but optimally in time $O(L)$. We can use this bulk synchrony to hide the synchronisation overhead, provided that we have sufficient parallel slackness so that each processor makes many memory accesses during each simulation step.

A common algorithm design technique is to have each processor take about L steps during each superstep, to balance communication and computation costs. Within each superstep each component sends or receives at most h messages in time T, then we can fix L as greater than T. Not surprisingly, many PRAM algorithms can be restructured specifically to provide for bulk synchrony using parallel slackness.

6.5 Memory management

To design algorithms for a virtual shared memory model machine, we need to know how to distribute the logical memory addresses among the physical locations of the machine such that distribution will not slow down the computation (i.e avoid contention and congestion). The distribution can be done randomly or deterministically.

Under fairly general assumptions, Upfal and Wigderson [156]) showed that an on-line simulation of T PRAM steps by a synchronous practical PRAM with the same number of processors requires $\Omega(T \log n / \log \log n)$ time, where n is the number of processors. Their simulation assumes each processing element of the practical PRAM consists of a computing element and a memory module, and the processing elements are connected by a complete network.

We saw in section 4.3.1 that the most promising method known for randomly evening out memory accesses (to avoid contention) is hashing. Using randomised hash functions, the simulation of a PRAM on a practical PRAM is governed by the following:

1. The time to evaluate the hash function.
2. The maximum number of shared memory accesses which are mapped to the same memory module under the hash function.
3. The time needed to access memory location (i.e communication latency l).

Simulation using hash functions was dealt with in [24, 74, 104, 129, 130, 157, 158]. Recently Karp and Luby [75] introduced a simulation which is more involved than that using a simple hashing scheme. The simulation uses two or more hash functions, and thus makes the contents of each PRAM cell accessible in two or more places.

As we saw in section 6.3, efficiency can be enhanced by using spatial locality of reference. The results of [24] show the possibility of exploiting locality during the simulations by using locality-preserving hash functions. This simulation supports block pipelining. Valiant [157, 158] gave strong theoretical evidence for supporting arbitrary pipelining during simulations of PRAM on the BSP model. Each processing element of the BSP model consists of a computational element and a memory module. Each computational element has a capability for efficiently computing hash addresses. This can be done by a hardware hashing module associated with a router interface without slowing down the computations performed by the processor.

Theorem 6.5.1 *[157, 158] Any EREW PRAM step of the v processors can be simulated on the p-processors BSP in optimal expected time ($O(v/p)$ time), provided $v = p \log p$ and $g = O(1)$, where g is the throughput of the router.*

Proof We randomly choose an appropriate hash function that will randomise memory and distribute memory requests evenly among p memory modules of the BSP. We distribute the v processors of a EREW PRAM so that each processor of the BSP model simulates $v/p = \log p$ of these. In one superstep the BSP model completes one EREW PRAM step. In the superstep each processor may need to access $\log p$ memory locations. Recall theorem 4.3.1. The expected largest number of accesses made to any memory module is $O(\log p)$. So each processor will send $\log p$ requests and each memory module will receive at most $O(\log p)$. Hence the duration of the superstep, L, needs to be large enough to accommodate the routing of a $\log p$-*relation*. From theorem 4.3.6, we can choose L as large as $O(g \log p)$ for the hypercube. By assuming $g = O(1)$, the superstep can be completed in optimal time.

□

Note that, if hashing is to be exploited efficiently, then the periodicity L may as well be at least logarithmic. Moreover, Valiant's results are justified only if we assume that the hash function can be evaluated in constant time.

The prevailing vision of general-purpose parallel computers is that the network topology should be hidden, but the programmer should retain control of memory management, including the decision whether or not to hash the shared memory. This decision should take into account the cost of hashing, as well as the relative intricacies of the model and simulated PRAM algorithms.

When the patterns of shared memory accesses are known in advance, the memory locations can be deterministically addressed, to avoid contention. This reduces the amount of slack required in programming [48, 157]. Moreover, we need not maintain logarithmic periodicity, i.e the length of a superstep does not need to be logarithmic.

6.6 Conclusion

Overall the material of this chapter demonstrates that there is no theoretical hindrance in designing massively parallel machines for parallel computation.

Chapter 7

Bulk Synchronous Parallel Algorithms

The work of this chapter was carried out as part of the ESPRIT I project PUMA as explained in the beginning of the last chapter and appeared in a report [132] to the European Commission. This chapter shows that scalable *transportable algorithms* can be written for certain basic tasks, balanced tree computations, Fast Fourier Transform (FFT) and matrix multiplications.

7.1 Introduction

There are two modes of programming, *automatic mode* and *direct mode*. In the automatic mode the virtual shared memory is distributed among memory modules by a hash function. Thus, the memory distribution is hidden from the user (as in the $PRAM$ model). However, optimal simulation with the automatic mode requires the slackness to be at least logarithmic, and g to be close to unity (theorem 6.5.1). Moreover, we assume that the hash function can be evaluated in constant time. If the patterns of shared memory accesses are known in advance, then the programmer can retain control of the memory management to avoid hashing. This is called the direct mode of programming. In this chapter we describe some transportable algorithms in the direct mode on a BSP model.

Recall that the BSP model is defined as the combination of four attributes [157]):

- A number of components each performing computational and/or memory operations.
- A common hashing function being evaluated by an individual hashing module associated with each computational element. Each hashing module is implemented in hardware.
- A router for routing data between computational and memory elements. The router operates independently of continued computation and storage access in the computational and memory elements.
- A synchroniser for all or a subset of the components.

A computation on the BSP model consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and message arrivals from other memory elements. After each period of L time units, a global check is made to determine whether the superstep has been completed. If it has, the machine proceeds to the next superstep. Otherwise the next period of L units is allocated to the unfinished superstep. The performance of a BSP algorithm measured by three parameters, p the number of processors, g a parameter such that an h-relation can be realised in gh steps, L which captures the minimum reasonable interval between global synchronisation. We can choose L as small as the time units to realise a μ-relation, where $\mu = L/g$. For example, if the network is a p-node hypercube then we choose $\mu = \log p/g$ and $L = \log p$.

Note that, the algorithms described below assume that each processor of the BSP model is performing computational and memory operations.

7.2 Balanced tree computation

As we saw in section 2.3.1, the complete binary tree computation can be performed on a EREW PRAM by p processors in $O(\log p)$ time. For the BSP model, when there is substantial communication latency, l, for example if $l = \log_2 p$, then the complete binary tree is no longer the natural structure for parallel computation. This is because the synchroniser needs to synchronise at the every level of the tree, and each superstep consists of 2 computational steps and each computational element sends or receives at most 2 messages. Recall that, each internal node of a complete binary tree has two children. Each superstep will take $O(\log p)$ time, that is $L = O(\log p)$. But each superstep consists of only 2 ($\ll L$) computational operations.

However, this can be improved by using a μ-ary tree [117, 48]. Figure 7.1 shows the 4-ary tree with 64 leaves. At each level of the tree, each active processor reads

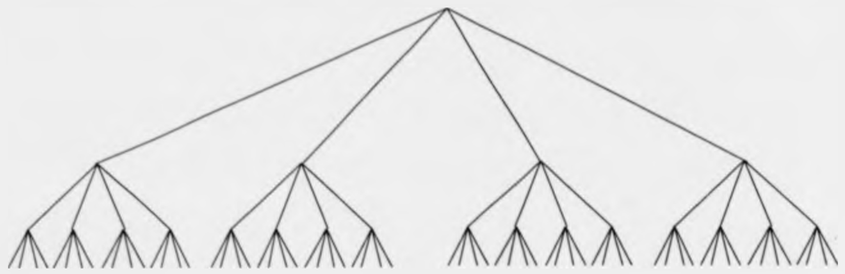


Figure 7.1:

μ values, computes at most L operations, writes the result, and then synchronises. Hence, if $\mu = p^{1/k}$ then k supersteps will suffice and each superstep will take $O(L)$ time.

7.3 Fast Fourier Transform

It is a well known fact that the FFT can be efficiently computed in parallel using a communication pattern that is a butterfly graph. Recall that, the v input butterfly has n rows and $\log v + 1$ levels (or columns). The inputs are at level 0 and the outputs at level $\log v$. By assigning a processor to each row we can simulate one level at a time. In this way a PRAM can compute the FFT in $O(\log v)$ time using v processors. Likewise, the BSP can simulate one such level at a time, synchronising after each layer, to solve an FFT problem in $\log p$ supersteps with p processors, where p is equal to v. Notice that in each superstep we need to realise a *1-relation*. If the time to realise *1-relation* is equal to μ-relation then we can improve upon the work as follows [55, 117].

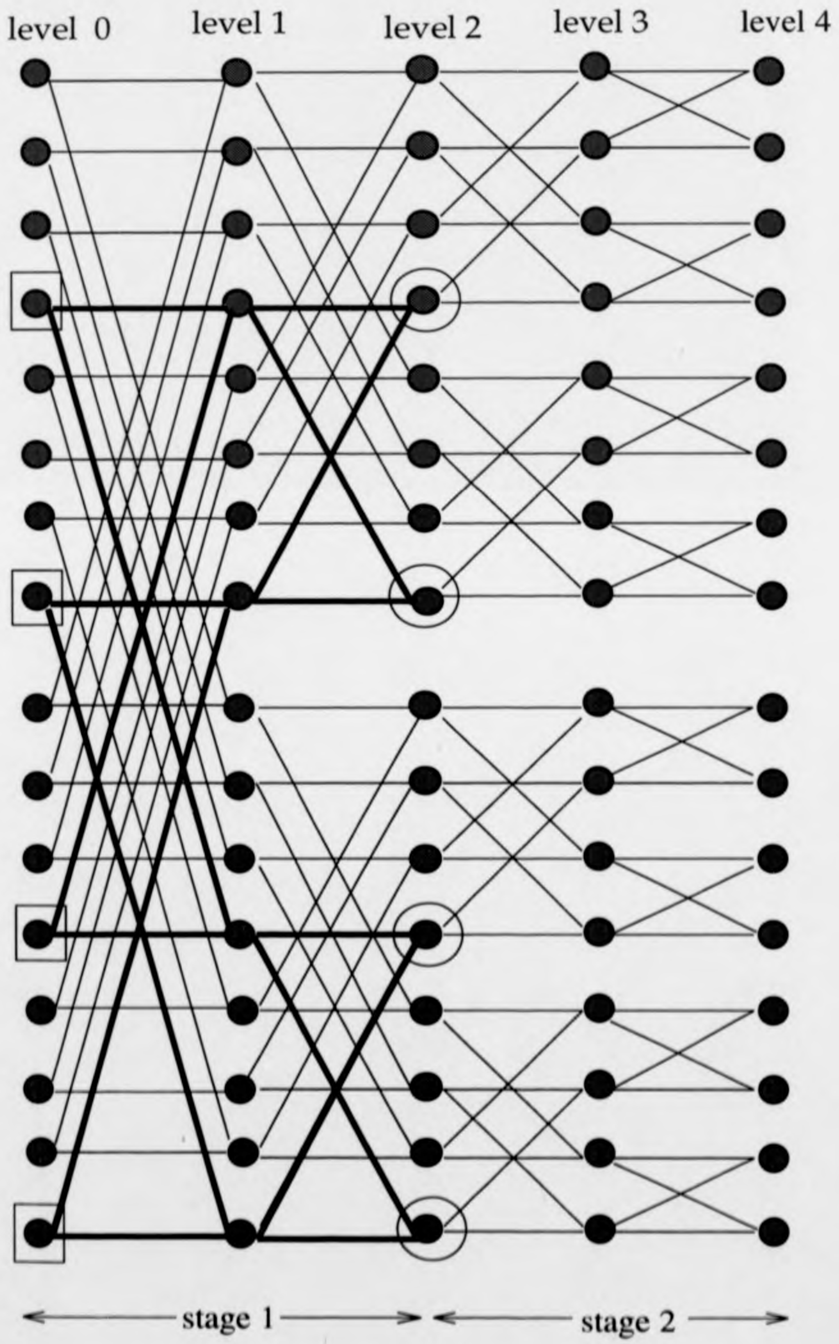


Figure 7.2:

We partition the levels of the butterfly into $\log v / \log \mu$ stages of $\log \mu$ consecutive levels each. By the structure of the butterfly, the value at each node in the last level of its stage depends on the values at μ nodes in the last level of the previous stage. Moreover, the values of a set of μ nodes in the last level of their stage depend on the values of a set of μ nodes in the last level of their previous stage. For example, figure 7.2 shows a butterfly graph of $v = 16$ inputs. The levels are divided into stages of $\log \mu$ levels when $\mu = 4$. The set of values of four circled nodes in last level of stage 1 depend on the set of values of four squared nodes in level 0. The dependencies between the circled nodes and the squared nodes are shown in thick edges. Interestingly, these thick edges form a butterfly graph of four inputs. There are four independent similar butterfly graphs in stage 1. In general, each stage consists of v/μ independent butterfly graphs of μ inputs each. Here the expressions are integers rounded appropriately. At each stage a processor can mimic each of these butterfly graphs of μ rows and $\log \mu$ levels in $\mu \log \mu$ sequential time. All the butterfly graphs in a stage can be executed in $\mu \log \mu$ time using v/μ processors. Once a stage (or superstep) is completed the next stage (or superstep) can start. In each superstep each processor computes $\mu \log \mu$ local operations and sends and receives μ messages. Each superstep will take $L = O(\mu \log \mu)$ time. Initially we distribute v inputs uniformly among $p = v/\mu$ processors, μ inputs in each. We can, therefore, evaluate the FFT on v/μ processors in $\log v / \log \mu$ supersteps, with total time $\mu \log v$. Note that the work of this algorithm is $O(v \log v)$ and equal to the work of the best known PRAM algorithm. Hence we have an optimal implementation.

7.4 Transitive closure and graph algorithms

As described in section 2.5.4, the transitive closure of an adjacency matrix solves several graph problems including topological sorting, strong components and all pairs shortest paths. Recall that the transitive closure A^* of an $n \times n$ matrix A is equal to $D^{2^{\lceil \log_2 n \rceil}}$, where $D = I \oplus A$ and I is the identity matrix. Here the matrix product is defined with two binary operators $\oplus$ and $\otimes$.

Suppose we have $p < n^2$ processors. We assume that the elements of A are initially distributed as evenly as possible among the p processors. First we compute the matrix D, this can be done in $O(1)$ time. Now A^* can be computed by repeated squaring of D as was explained in section 2.3.2, chapter 2. Each squaring operation is performed as follows. This is similar to the algorithm of Valiant [157] for computing the product of two $n \times n$ matrices.

Consider the s^{th} squaring operation on matrix D, i.e. computing D^{2s} from D^s, for $1 \leq s \leq \lceil \log_2 n \rceil$. We assign to each processor the sub problem of computing an $(n/\sqrt{p} \times n/\sqrt{p})$ submatrix of D^{2s}. To compute the $(i,j)^{th}$ position of D^{2s}, a processor has to receive the i^{th} row and the j^{th} column of D^s. Suppose a processor computes positions (k, j) of D^{2s}, where $l \leq j < l + n\sqrt{p}$. Then the processor has to receive data describing the k^{th} row and columns from the l^{th} column to the $((l + n\sqrt{p}) - 1)^{th}$ column of matrix D^s. That is, the processor has to receive n elements of the row and $n(n/\sqrt{p})$ elements of the column. There are $n/\sqrt{p}$ such rows required to compute all the positions of the $n/\sqrt{p} \times n/\sqrt{p}$ submatrix. Thus, each processing elements has to receive $2n^2/\sqrt{p}$ elements of D^s. Now consider the number of messages each processor has to send. Elements of each row of

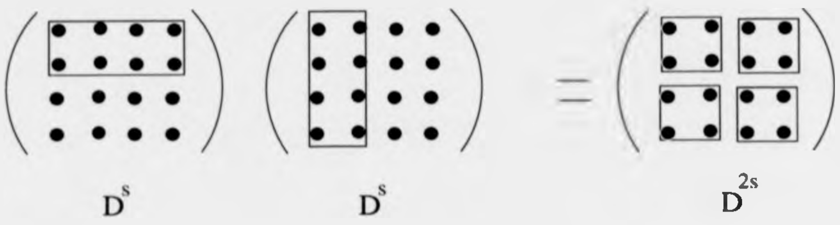


Figure 7.3:

D^s is required by $\sqrt{p}$ processing elements. This is because each row of D^{2s} is partitioned into $\sqrt{p}$ pieces of length $n/\sqrt{p}$, and the pieces are computed by distinct processing elements. Similarly, elements of each column of D^s required by $\sqrt{p}$ processing elements. Thus each element of D^s is required by $2\sqrt{p}$ processors. Assume that the n^2 elements describing D^s are distributed uniformly among p processors. Each processor has n^2/p elements of D^s. Each processor thus has to replicate each of its elements $2\sqrt{p}$ times and send the appropriate elements to the $2\sqrt{p}$ processors requiring them. Hence we have a communication pattern that each processor has to send $2n^2/\sqrt{p}$ messages and receive $2n^2/\sqrt{p}$ messages, i.e. realising $2n^2/\sqrt{p}$-relation. Now each processor can compute an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of D^{2s} in $2n^3/p$ sequential time, using the standard sequential algorithm. Suppose $2n^2/\sqrt{p} = \mu$, then we can square the matrix D^s in $O(n^3/p)$ time (i.e. in a superstep of length $O(n^3/p)$). For example, figure 7.3 shows the partition of D^{2s} when $n^2 = 16$ and $p = 4$. The elements of the matrices are shown in thick dots. Each square in D^{2s} represents a subproblem. Each processor computes the elements in a square in D^{2s}. For example, the processor computes the elements of the top left hand corner square has to receive the elements of D^s. These elements are enclosed in squares on the left hand side of the figure.

Notice that for the next squaring operation on D^{2^n} to compute D^{4^n}, the elements of D^{2^n} are distributed uniformly among p processors by the squared operation described above. We can compute the transitive closure in $\lceil \log_2 n \rceil$ supersteps of length n^3/p using p processors. The total work of this algorithm is $O(n^3 \log n)$. We thus have an optimal parallel algorithm for topological sorting, strong components and all pairs shortest paths on graphs.

7.5 Further work

One can observe from this chapter that algorithmic design requires a new discipline to get optimal algorithms in the BSP model. A systematic study ought to be the subject of an extensive research program. Some work towards this direction is described in [48].

Chapter 8

Conclusions

This thesis reviewed the evidence for the statement[166] that: *Unless parallel machines are designed to support the PRAM, or a model of parallel computation which is very close to it, the design of parallel algorithms is doomed to be a very difficult (or even impossible) task.* In the course of this review we presented new results concerned with parallel approximation algorithms (chapter 3), embeddings (chapter 5) and the design of bulk synchronous algorithm (chapter 7). This work has appeared in the literature and has been reported at conferences as detailed in the declaration at the beginning of this thesis.

We have seen that the PRAM provides a very simple and natural architecture independent model for the parallel algorithm designer. Furthermore, the PRAM has proved to be a valuable tool for theoretical computer scientists studying the power and fundamental limitation of parallelism. Unfortunately, the *gap* between real parallel machines and the PRAM may force us to think that the PRAM is not a particularly practical model for general purpose parallel computing. However, the theoretical community has proved that the *gap* between the PRAM and feasible parallel models can be bridged. Solutions have been found for effectively interconnecting processing elements, for routing data on these networks and for distributing the data among memory modules without hotspots. Using these solutions, we have reviewed the possibility of general purpose computing employing a bridging model. Such a model acts as an interface between the actual hardware and the PRAM model. We reviewed the evidence that if a practical model can be viewed as a PRAM by the user (i.e the model hides all the hardware details) then this will achieve scalable parallel performance and portable parallel software. We demonstrated that PRAM algorithms can be optimally implemented on such practical models.

Chapter 2 described algorithmic tools and techniques which have been frequently used to place many problems in the class NC. In particular, we saw that by computing the transitive closure (of the adjacency matrix) several graph problems can be placed in NC. Unfortunately, this technique does not lead to an efficient parallel algorithm. At this time no efficient parallel solutions are known for this problem. This difficulty is known as the *transitive closure bottleneck* [128]. Designing efficient algorithms without the transitive closure technique for those problems is a challenging area for research.

In chapter 3 we considered the notion of parallel approximation algorithms. In particular we provide such an algorithm for finding minimum weight perfect matching. The question of whether the problem of finding an exact solution to the minimum-weight perfect matching problem can be placed in the class NC remains open. Resolution of the existence or otherwise of appropriate algorithms in this area may ultimately help to place more precise boundaries around what ought to be regarded as *tractable* problems for parallel computation. The minimum-weight perfect matching problem is still open for complete weighted graphs and even if they satisfy the triangle inequality. The algorithm of chapter 3 places the problem of finding an approximate minimum-weight perfect matching in a complete weighted graph satisfying the triangle inequality in NC with a performance ratio of $2\log_3 n$. The algorithm is conceptually very simple and comes within a $\log^2 n$ factor of the work measure of the sequential algorithms. It is also the first NC-approximation algorithm for the task with a sub-linear performance ratio.

In chapter 5 we describe dense edge-disjoint embeddings of the complete binary tree with n leaves in the following n-node communication networks: the hypercube, the de Bruijn and shuffle-exchange networks and the 2-dimensional mesh. In the embeddings the maximum distance from a leaf to the root of the tree is asymptotically optimal. The embeddings facilitate efficient implementation of many PRAM algorithms on these networks. Note that this technique is architecture dependent. However, embedding may be hidden by system software/hardware in due course.

In chapter 6 we reviewed the practical PRAM models (in particular the BSP model) for architecture independent parallel algorithm design. These models differ from the well studied PRAM in two important parameters namely l and g. Although the models can cope with the current best values of l and g, we suggest to continue to improve these values. A desirable goal is to obtain values of the same order as for the PRAM ($l = g = O(1)$).

In chapter 7 we described some (*direct*) bulk synchronous algorithms, but a systematic study ought to be the subject of an extensive research program. Some work towards this direction is described in [48].

We described some results concerned with fault-tolerant, for example fault-tolerant routing using the IDA. However, in this thesis we have not given much attention to the problem of coping with processor failures. This is important for large parallel systems. The larger the number of processors, the greater the probability of failure. But, we assumed that the processing elements of the BSP model operate correctly at all times. In this context efficient techiques that will allow PRAM algorithms to run optimally on fault-prone practical PRAMs need to be developed. Some important work has already been done in [78, 79, 80]. However, coping with processor failures of a parallel model with communication delay remains to be done.

Bibliography

[1] A. Aggarwal, A.K. Chandra and M. Snir, "Communication complexity of PRAMs", *Theoretical Computer Science*, Vol. 71, 3-28, 1990.

[2] A. Aggarwal, A.K. Chandra and M. Snir, "On communication latency in PRAM computations", *Symposium on Parallel Architectures and Algorithms*, 11-21, 1989.

[3] A. V. Aho, J.E. Hopcroft, and J.D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974

[4] W. Aiello, T. Leighton, B. Maggs and M. Newman, "Fast algorithms for bit-serial routing on a hypercube", in *Proceedings of* 2nd *Annual ACM Symposium on Parallel Algorithms and Architectures*, 55-64, 1990.

[5] M. Ajtai, J. Komlos and E. Szemeredi, "Sorting in $c \log n$ steps", *Combinatorica*, Vol. 3, 1-19, 1983.

[6] S.G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice-Hall International Editions, 1989.

[7] R. Aleliunas, "Randomised parallel communication", in *Proceedings of the*

ACM Symposium on Principles of Distributed Computing, 60-72, 1982.

[8] R.J. Anderson and G.L. Miller, "Optical communication for pointer based algorithms", *Technical Report CRI 88-14*, University of Southern California, 1988.

[9] R. Anderson and E. Mayr, "A P-complete problem and approximation to it", *Research Report STAN-CS-84-1014*, Department of Computer Science, Stanford University, 1984.

[10] E. Arjmandi, M.J. Fischer and N.A. Lynch, "Efficiency of synchronous versus asynchronous systems", *Journal of the ACM*, Vol. 30 No. 3, 449-456, 1983.

[11] J. Aspens and M. Herlihy, "Wait free data structures in the asynchronous PRAM model", *Journal of the ACM*, 340-349, 1990.

[12] Y. Aumann and A. Schuster, "Deterministic PRAM simulation with constant memory blow-up and no time-stamps", in *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, 22-29, 1990.

[13] K. Batcher, "Sorting networks and their applications", in *Proceedings of the AFIPS Spring Joint Computing Conference*, Vol.32, 307-314, 1968.

[14] P. Became and J. Hastad, "Optimal bounds for decision problems on the CRCW PRAM", *In Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 83 -93, 1987.

[15] V. Benes, "Permutation groups, complexes and rearrangeable multistage connecting networks", *Bell System Technical Journal*, Vol. 43, 1619-1640, 1964.

[16] S.N. Bhatt and I.C.F. Ipsen, "How to embed trees in hypercubes", *Research Report Yale/DCS/RR-443*, Department of Computer Science, Yale University, 1985.

[17] G. Blelloch, "Scans as primitive parallel operations", in *Proceedings of International Conference on Parallel Processing*, 355-362, 1987.

[18] A. Borodin, "On relating time and space to size and depth", *SIAM Journal on Computing*, Vol. 6, 733-744, 1977.

[19] A. Borodin and J.E. Hopcroft, "Routing, merging and sorting on parallel model of computation", *Journal of Computer System Science*, Vol. 30, 130-145, 1985.

[20] R.P. Brent, "The parallel evaluation of general arithmetic expressions", *Journal of the ACM*, Vol.21, 201-206, 1974.

[21] J. I. Carter and M. N. Wegman, "Universal classes of hash functions", *Journal of Computer and System Sciences*, Vol. 18, 143-154, 1979.

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

[23] K. M. Chandy and S. Taylor, *An introduction to parallel programming*, Jones and Bartlett Publishers, Boston, 1992

[24] A. Chin, *Complexity issues in general purpose parallel computing*, D. Phil. thesis, University of Oxford, Oxford, 1991.

[25] F.Y. Chin, J. Lam and I. Chen, "Efficient parallel algorithms for some graph problems", *Communications of the ACM*, Vol. 25, 659-665, 1982.

[26] R. Cole and U. Vishkin, "Approximation and exact parallel scheduling with applications to list, tree and graph problems", in *Proceedings of* 27th *Annual IEEE Symposium on Foundations of Computer Science*, 32-53, 1986.

[27] R. Cole and U. Vishkin, "Approximate parallel scheduling. Part1: The basic technique with applications to optimal parallel list ranking in logarithmic time", *SIAM Journal on Computing*, Vol. 17, 128-142, 1988.

[28] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking", *Information and Control*, Vol. 81, 335-352, 1989.

[29] R. Cole and U. Vishkin, "Optimal parallel algorithms for expression tree evaluation and list ranking", in *Proceedings of Aegean Workshop on Computing*, 91-100, 1988.

[30] R. Cole and U. Vishkin, "Deterministic coin tossing with applications to optimal parallel list ranking", *Information and Control*, Vol. 70, 32-53, 1986.

[31] R. Cole and U.Vishkin, "Deterministic coin tossing and accelerating cascades: micro and macro techniques for sesigining parallel algorithm", in *Proceedings of* 18th *Annual ACM Symposium on Theory of Computing*, 206-219, 1986.

[32] R. Cole and O. Zajicek, "The APRAM: Incorporating asynchrony into the PRAM model", in *Proceedings of* 1st *Annual ACM Symposium on Parallel Algorithms and Architectures*, 169-178, 1989.

[33] R. Cole and O. Zajicek, "The expected advantage of asynchrony", in *Proceedings of* 2nd *Annual ACM Symposium on Parallel Algorithms and Architectures*, 85-94, 1990.

[34] S.A. Cook, C. Dwork and R. Reischuk, "Upper and lower time bounds for parallel random access machines without simultaneous writes", *SIAM Journal on Computing*, Vol. 15, 87-97, 1986.

[35] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions", *Proceedings of* 19th *Annual ACM Symposium on Theory of Computing*, 1-6, 1987.

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

[37] E. Dahlhaus and M. Karpinski, "Parallel Construction of Perfect Matching and Hamiltonian Cycles on Dense Graphs", *Theoretical Computer Science*, Vol. 61, 121-136, 1988.

[38] C. O Dunlaing, "Some parallel geometric algorithms", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 77-108, 1993.

[39] P.E. Dunne, "A result on k-valent graphs and its application to a graph embedding problem", *Acta Informatica*, Vol. 24, 447-459, 1987.

[40] J. Edmonds, "Matching and Polyhedrons with 0,1 Vertices", *Journal of Research of the National Bureau of Standards B*, 125-130, 1965.

[41] J. Edmonds, "Paths, trees and flowers", *Canadian Journal of Mathematics*, Vol. 17, 449-467, 1965.

[42] D. Eppistein and Z. Galil, "Parallel algorithmic techniques for combinatorial computation", *Annual Review of Computer Science*, 233-283, 1988.

[43] S. Fortune and J. Wyllie, "Parallelism in random access machines", *Proceedings of* 10^{th} *Annual ACM Symposium on Theory of Computing*, 114-118, 1978.

[44] H.N. Gabow, *Implementations of algorithm for Maximum Matching on Nonbipartite Graphs*, Ph.D. Dissertation, Department of Computer Science, Stanford University, 1974.

[45] Z. Galil and V. Pan, "Improved Processor Bounds for Combinatorial Problems in RNC", *Combinatorica*, Vol. 8, 189-200, 1988.

[46] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, Freeman, 1979.

[47] H. Gazit, G. L. Miller and S. H. Teng, "Optimal tree contraction in the EREW model", in *Proceedings of Concurrent Computations: Algorithms, Architecture and Technology*, Plenum, New York, 139-156, 1988.

[48] A.V. Gerbessiotis and L.G. Valiant, "Direct bulk-synchronous parallel algorithms", *Technical Report TR-10-92*, Center for Research in Computing Technology, Harvard University, 1992.

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

[50] A. M. Gibbons, "An introduction to distributed memory models of parallel computation", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 197-215, 1993.

[51] A.M. Gibbons and M.S. Paterson, "Dense edge-disjoint embedding of binary trees in the mesh", in *Proceedings of the* 4th *Annual ACM Symposium on Parallel Algorithms and Architectures*, 257-263, 1992.

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

[53] A. M. Gibbons and W. Rytter, "An optimal parallel algorithm for dynamic expression evaluation and its application", in *Proceedings of Symposium on Foundations of Software Technology and Theoretical Computer Science*, 453-469, 1986.

[54] A. M. Gibbons and R. Ziani, "The balanced binary tree technique on mesh connected computers", *Information Processing Letters*, Vol. 37, 101-109, 1991.

[55] P. B. Gibbons, *The asynchronous PRAM: a semi-synchronous model for shared memory MIMD machines*, Ph.D. Thesis, University of California at Berkeley, 1989.

[56] A.V. Goldberg, S.A. Plotkin and G.E. Shannon, "Parallel symmetry-breaking in sparse graphs", in *Proceedings of the* 19th *Annual ACM Symposium on Theory of Computing*, 1987.

[57] L.M. Goldschlager, "A unified approach to models of synchronous parallel machines", *Journal of the ACM*, Vol. 29, 1073-1086, 1982.

[58] A.Gottlieb, R. Grishman, C.P. Kruskal, K.P. Mcauliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer-designing an MIMD shared memory parallel computer", *IEEE Transactions on Computers*, Vol. C-32, 175-189, 1983.

[59] R. Greenlaw, H.J. Hoover and W.L. Ruzzo, "A Compendium of Problems Complete for P", *Technical Report TR 91-11*, Department of Computer Science, University of Alberta, 1991.

[60] D.Y. Grigoriev and M. Karpinski, "The Matching Problem for Bipartite Graphs with Polynomially Bounded Permanents is in NC", *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, 166-172, 1987.

[61] J. Hastad, T. Leighton and M. Newman, "Fast computation using faulty hypercubes", in *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, 251-263, 1989.

[62] X. He and Y. Yesha, "Binary tree algebraic computation and parallel algorithms for simple graphs", Journal of Algorithms, Vol. 9, 6-20, 1986.

[63] D. Hembold and E. Mayer, "Two-processor Scheduling is in NC", *VLSI Algorithm and Architectures*, editors: Makedon et al., Lecture Notes in Computer Science, Vol. 227, 12-25, 1986.

[64] S. W. Hornick and F. P. Preparata, "Deterministic P-RAM simulation with constant redundancy", in *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, 103-109, 1989.

[65] C.S. Iliopoulos, "Parallel algorithms for string matching", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 109-121, 1993.

[66] M. Iri, K. Murota and S. Matsui, "Linear-time Approximation Algorithms for Finding the Minimum-Weight Perfect Matching on a Plane", *Information*

Processing Letters, Vol. 12, 206-209, 1981.

[67] A. Israeli and Y. Shiloach, "An improved algorithm for maximal matching", *Information Processing Letters*", Vol. 33, 57-60, 1986.

[68] J. JaJa, *An Introduction to Parallel Algorithms*, Addison Wesley, 1992.

[69] M. R. Jerrum and S. Skyum, "Families of fixed degree graphs for processor 'interconnection", *IEEE Transaction on Computing*, Vol. 33, 190-194, 1984.

[70] D.B. Johnson and P. Metaxas, "Connected components in $O(\log^{3/2} | V |)$ parallel time for the CREW PRAM", in *Proceedings of the* 31st *Annual IEEE Symposium on Foundations of Computer Science*, 1991

[71] N. Kahale, "Better expansion for Ramanujam graphs", In *Proceedings of the* 32nd *Annual IEEE Symposium on Foundations of Computer Science*, 398-404, 1991.

[72] C. Kaklamanis, D. Krizanc and T. Tsantilas, "Tight bounds for oblivious routing in the hypercube", In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 31-36, 1990.

[73] D.R. Karger, N. Nisan and M. Parnas, "Fast Connected Components Algorithms for the EREW PRAM", *4th Annual ACM Symposium on Parallel Algorithms and Architectures*, 373-38, 1992.

[74] A. R. Karlin and E. Upfal, "Parallel hashing: An efficient implementation of shared memory", *Journal of the ACM*, Vol. 35, No. 4, 876-892, 1988.

[75] R.M. Karp and M. Luby, "Efficient PRAM simulation on a distributed memory machine", in *Proceedings of the* 24th *Annual ACM Symposium on Theory of Computing*, 318-326, 1992.

[76] R. Karp and V. Ramachandran, "Parallel algorithms for Shared Memory Machines", *Handbook of Theoretical Computer Science*, J. van Leeuwen (editor), Vol.1, Elsevier and MIT Press, 1991.

[77] R. Karp, E. Upfal and A. Wigderson, "Constructing a Perfect Matching is in Random NC", *Proceedings of the* 17th *Annual ACM Symposium on Theory of Computing*, 22-32, 1985.

[78] Z.M. Kedem, K.V. Palem, A. Raghunathan and P.G. Spirakis, "Combining tentative and definite executions for very fast dependable parallel computing", in *Proceedings of* 23rd *Annual ACM Symposium on Theory of Computing*, 381-390, 1991.

[79] Z.M. Kedem, K.V. Palem, A. Raghunathan and P.G. Spirakis, "Resilient parallel computing on unreliable parallel machines", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 149-175, 1993.

[80] Z.M. Kedem, K.V. Palem, and P.G. Spirakis, "Efficient robust parallel computations", in *Proceedings of* 22nd *Annual ACM Symposium on Theory of Computing*, 138-148, 1990.

[81] L. Kirousis, M. Serna and P. Spirakis, "The parallel complexity of the subgraph connectivity problem", in *Proceedings of the* 30th *Annual IEEE Symposium*

on Foundations of Computer Science, 294-299, 1989.

[82] L. Kirousis and P. Spirakis, "Probabilistic log-space reductions and problems probabilistically hard for P", in *Proceedings of the* 1st *Scandinavian Workshop on Algorithm Theory*, 1988.

[83] C. P. Kruskal, L. Rudolph and M. Snir, "A complexity theory of efficient parallel algorithms", *Theoretical computer science*, Vol. 71, 1990, 95-132.

[84] C. Kruskal and M. Snir, " A unified theory of interconnection network structure", *Theoretical Computer Science*, Vol. 48(1), 75-94, 1986.

[85] L. Kucera, "Parallel computation and conflicts in memory access", *Information Processing Letters*, Vol. 14, 93-96, 1982.

[86] R.E. Ladner and M.J. Fisher, Parallel prefix computation, *Journal of the ACM*, Vol. 27, 831-838, 1980.

[87] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt-Rinehart-Winston, New York, 1976.

[88] T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann, California, 1992.

[89] T. Leighton, "Methods for message routing in parallel machines", in *Proceedings of the* 24th *Annual ACM Symposium on Theory of Computing*, 77-96, 1992.

[90] T. Leighton and B. Maggs, "Expanders might be practical", in *Proceedings of the* 30th *Annual IEEE Symposium on Foundations of Computer Science*,

384-389, 1989.

[91] T. Leighton and C.G. Plaxton, "A (fairly) simple circuit that (usually) sorts", in *Proceedings of* 31st *Annual IEEE Symposium on Foundations of Computer Science*, 1990.

[92] C. E. Leiserson, "Fat trees: universal network for hardware efficient supercomputing", in *Proceedings of the International Conference on Parallel Processing*, 393-402, 1985.

[93] C.E. Leiserson, "Area-efficient graph layouts (for VLSI)", in *Proceedings of the* 21st *Annual IEEE Symposium on Foundations of Computer Science*, 270-281, 1980.

[94] L. Lovasz, "Computing ears and branching in parallel",in *Proceedings of* 26th *Annual IEEE Symposium on Foundation of Computer Science*, 464-467, 1985.

[95] A. Lubotzky, R. Phillips and P. Sarnak, "Ramanujan graphs", *Combinatorica*, Vol. 8, 261-277, 1988.

[96] N. A. Lynch and M. J. Fischer, "On describing the behavior and implementation of distributed systems", *Theoretical computer science*, Vol. 13, 17-43, 1981.

[97] Y.-D. Lyuu, "Fast fault-tolerant parallel communication and on-line maintenance using information dispersal", in *Proceeding of the* 2nd *Annual ACM Symposium on Parallel Algorithms and Architectures*, 378-387, 1990.

[98] Y.-D. Lyuu, "Fast fault-tolerant parallel communication with law congestion and on-line maintenance using information dispersal", *Technical Report TR-19-89*, Aiken Computation Lab., Harvard University, 1989.

[99] E.S. Maniloff, K.M. Johnson and J. Reif, "Holographic routing network for shared-memory parallel computers", *Technical Report CSE-89-8*, University of California at Davis, 1989.

[100] Y. Maon, B. Schieber and U. Vishkin, "Parallel ear decomposition search (EDS) and st-numbering in graphs", *Theoretical Computer Science*, Vol. 47, 277-298, 1986.

[101] C. Martel, A. Park and R. Subramonian, "Optimal asynchronous algorithms for shared memory parallel computers", *Technical Report CSE-89-8*, University of California at Davis, 1989.

[102] E.W. Mayr, "Parallel Approximation Algorithms", *Research Report*, Department of Computer Science, Stanford University, California.

[103] W.F. McColl, "General purpose parallel computing", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.), Cambridge University Press, 243-296, 1993.

[104] K. Mehlhorn and U. Vishkin, "Randomised and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories", *Acta Informatica*, Vol. 21, 339-374, 1984.

[105] G. L. Miller and J. H. Reif, "Parallel tree contraction and its application", in *Proceedings of the* 19th *Annual ACM Symposium on Theory of Computing*, 254-263, 1987.

[106] J. Misra, "Phase synchronization", *Information Processing Letters*, Vol. 38, 101-105, 1991.

[107] S. Miyano, S. Shiraishi and T. Shoudai, "A List of P-Complete problems", *Technical Report RIFIS-TR-CS-17*, Research Institute of Fundamental Information Science, Kyushu University, Japan, 1989.

[108] B. Monien and H. Sudborough, "Comparing interconnection networks", in *Proceedings of the Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 324, 139-153, Springer-Verlag, 1988.

[109] J.K. Mullin, "A caution on universal classes of hash functions", *Information Processing Letters*, Vol. 37, 247-256, 1991.

[110] K. Mulmuley, U. Vazirani and V. Vazirani, "Matching is as easy as matrix inversion", in *Proceedings of the* 19^{th} *Annual ACM Symposium on Theory of Computing*, 345-354, 1987.

[111] J. Naor, "Computing a Perfect Matching in a Line Graph", in *Proceedings of the* 9^{th} *Conference on the Foundations of Software Technology and Theoretical Computer Science*, 139-148, 1989.

[112] D. Nassimi and S. Sahni, "An optimal routing algorithm for mesh connected computers", *Journal of the ACM*, Vol. 27, 6-29, 1980.

[113] N. Nishimura, "Asynchronous shared memory parallel computation", in *Proceedings of the* 2^{nd} *Annual ACM Symposium on Parallel Algorithms and Architectures*, 76-84, 1990.

[114] D. Nussbaum and A. Aggarwal, "Scalability of parallel machines", *Communication of the ACM*, Vol. 34, 57-61, 1991.

[115] C.N.K. Osiakwan and S.G. Akl, "The Maximum weight perfect vmatching problem for complete weighted graphs is in PC", *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, 880-887, 1990.

[116] H. Papadimitriou and J. D. Ullman, "A communication-time tradeoff", *SIAM Journal on Computing*, Vol. 16, 260-269, 1984.

[117] H. Papadimitriou and M. Yannakakis, "Towards an architecture-independent analysis of parallel algorithms", in *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 510-513, 1988.

[118] M.S. Paterson, W.L. Ruzzo, L. Snyder, "Bounds on minimax edge-length for complete binary trees", in *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, 293-299, 1981.

[119] G.F. Pfister and V.A. Norton, "Hot spot contention and combining in multistage interconnection networks", *IEEE Transactions on Computers* , Vol. C-34, No. 10, 943-948,1985.

[120] N. Pippenger, "Parallel communication with limited buffers", in *Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science*, 127-136, 1984.

[121] D.A. Plaisted, "Heuristic Matching for Graphs Satisfying the Triangle Inequality", *Journal of Algorithms*, Vol. 5,163-179, 1984.

[122] V.R. Pratt and L.J. Stockmeyer, "A characterization of the power of vector machines", *Journal of Computer System Science*, Vol. 12, 198-221, 1976.

[123] F. Preparata and J. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation", *Communication of the ACM*, Vol.24(5), 300-309, 1981.

[124] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill International Editions, 1988.

[125] M.O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance", *Journal of the ACM*, Vol. 36, 335-348, 1989.

[126] P. Ragde, "Analysis of an asynchronous PRAM algorithm", *Information Processing Letters*, Vol. 39, 253-256, 1991.

[127] S. Rajasekaran and J. H. Reif, "Optimal and sublogarithmic time randomised parallel sorting algorithms", *SIAM Journal on Computing*, 594-607, 1989.

[128] V. Ramachandran, "Efficient parallel graph algorithms", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 67-76, 1993.

[129] A. G. Ranade, *The fluent abstract machine*, Ph. D. Thesis, Yale University, 1989.

[130] A. G. Ranade, "How to emulate shared memory", in *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, 185-194, 1987.

[131] S.B. Rao, "Properties of an interconnection architecture based on wavelength division multiplexing", *TR-92-009-3-0054-2*, NEC Research Institute, Princeton, 1992.

[132] S. Ravindran, B. Dessau and A.M. Gibbons, "An overview of PRAM to practical PRAM algorithmics", *Report 6.2.1, Parallel Universal Message-passing Architecture*, ESPRIT Project 2701 of the EC, 1991.

[133] S. Ravindran and A.M. Gibbons, "Dense edge-disjoint embedding of binary trees in the hypercube", *Information Processing Letters*, Vol. 45, 321-325, 1993.

[134] S. Ravindran and A.M. Gibbons, "Densely embedding the complete binary tree in communication networks", 9th *British Colloquium for Theoretical Computer Science*, University of York, England, March 1993.

[135] S. Ravindran, A.M. Gibbons and M.S. Paterson, "Dense edge-disjoint embedding of complete binary trees in interconnection networks", to appear in *Theoretical Computer Science*, 1994.

[136] S. Ravindran, N.W. Holloway and A.M. Gibbons, "Approximating minimum weight perfect matchings, for complete graphs satisfying the triangle inequality", in *Proceedings of* 19th *International Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, Springer-Verlag, 1993, To appear.

[137] E.M. Reingold and R.E. Tarjan, "In a greedy heuristic for complete matching", *SIAM Journal on Computing*, Vol. 10, 676-681, 1981.

[138] J. H. Reif and L. G. Valiant, "A logarithmic time sort for linear size networks", *Journal of the ACM*, Vol. 34, 60-76, 1987.

[139] W. L. Ruzzo, "On uniform circuit complexity", *Journal of Computer and Systems Sciences*, Vol. 22, 365-383, 1981.

[140] B. Schieber and U. Vishkin, "On finding lowest common ancestors: simplification and paralization", *SIAM Journal on Computing*, Vol. 17, 1253-1262, 1988.

[141] M.J. Serna, *The Parallel Approximability of P-complete Problems*, Ph.D. Thesis, Dep. de Llenguatges I Sistmes Informatics, Universitat Politecnica de Catalunya, 1990.

[142] M.J. Serna and P. Spirakis, "The approximability of problems complete for *P*", in *Proceedings of the International Symposium on Optimal Algorithms*, Lecture Notes in Computer Science, Springer-Verlag, Vol. 401, 193-204, 1989.

[143] D. Shasha and M. Snir, "Efficient and correct execution of parallel programs that share memory", *ACM Transactions on Programming Languages and Systems*, Vol. 10, 282-312, 1988.

[144] Y. Shiloach and U. Vishkin, "An O(log n) parallel connectivity algorithm", *Journal of Algorithms*, Vol. 3, 57-67, 1982.

[145] A. Siegel, "On universal classes of fast high-performance hash functions, their time-space tradeoff, and their applications", in *Proceedings of the* 30th *Annual IEEE Symposium on Foundations of Computer Science*, 20-25, 1989.

[146] H. Siegel, "Interconnection networks for SIMD machines", *Computer*, Vol. 12(6), 57-65, 1979.

[147] H. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, Lexington Books, Lexington, MA, 1984.

[148] P. Spirakis, "PRAM models and fundamental parallel algorithmic techniques: part II (randomized algorithms)", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 77-108, 1993.

[149] L. Stockmeyer and U. Vishkin, "Simulations of Parallel Random Access Machines by Circuits", *SIAM Journal on Computing*, Vol. 13, 409-422, 1984.

[150] H. Stone, "Parallel processing with perfect shuffle", *IEEE Transactions on Computers*, Vol. C-20, 153-161, 1971.

[151] K.J. Supowit, D.A. Plaisted and E.M. Reingold, "Heuristics for weighted perfect matching", *Proceedings of the* 12th *Annual ACM Symposium on Theory of Computing*, 398-419, 1980.

[152] R.E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time", in *Proceedings of the* 25th *Annual IEEE Symposium on the Foundations of Computer Science*, 12-20, 1984.

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

[154] E. Upfal, "Efficient schemes for parallel communication", *Journal of the ACM*, Vol. 31, 507-517, 1984.

[155] E. Upfal, "An O(log n) deterministic packet routing scheme", *Journal of the ACM*, Vol.39, 55-70, 1992.

[156] E. Upfal and A. Wigderson, "How to share memory in a distributed system", *Journal of the ACM*, Vol.34, 116-127, 1987.

[157] L. G. Valiant, "A bridging model for parallel computation", *Communication of the ACM*, Vol.33, 103-111, 1990.

[158] L. G. Valiant, "General purpose parallel architectures", *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* (J. Van Leeuwen, ed.), Amsterdam: North-Holland, 944-971, 1990.

[159] L.G. Valiant, "A combining mechanism for parallel computers",*Technical Report TR-24-92*, Center for Research in Computing Technology, Harvard University, 1992.

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

[161] L.G. Valiant, "A scheme for past parallel communication", *SIAM Journal on Computing*, Vol. 11, 350-361, 1982.

[162] L.G. Valiant, "Universality considerations in VLSI circuits", *IEEE Transcations on Computing*, Vol. 30, 135-140, 1981.

[163] L.G. Valiant and G.J. Brebner, "Universal schemes for parallel communications", in *Proceeding of the* 13th *Annual Symposium on Theory of Computing*, 263-277, 1981.

[164] U. Vishkin, "Implementation of simultaneous memory address access in models that forbid it", *Journal of Algorithms*, 45-50, 1983.

[165] U. Vishkin, "A parallel-design distributed-implementation(PDDI) general-purpose computer", *Theoretical Computer Science*, Vol. 13, 157-172, 1984.

[166] U. Vishkin, "Structural parallel algorithmics", in *Lectures in Parallel Computation* (A.M. Gibbons and P.G. Spirakis, eds.,), Cambridge University Press, 1-18, 1993.

[167] U. Vishkin, "Randomized speed-ups in parallel computations", in *Proceedings of the* 16^{th} *Annual ACM Symposium on Theory of Computing*, 230-239, 1984.

[168] A. Waksman, "A permutation network", *Journal of the ACM*, Vol. 15, 159-163, 1968.

[169] J. C. Wyllie, *The complexity of parallel computation*, Ph. D. thesis, Department of Computer Science, Cornell University, 1979