Deterministic Simulation of Shared Memory on Bounded Degree Networks

Kieran T. Herley
Ph.D Thesis

TR 90-1090
February 1990

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

DETERMINISTIC SIMULATIONS OF SHARED MEMORY ON BOUNDED DEGREE NETWORKS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Kieran T. Herley

January 1990

© Kieran T. Herley 1990

ALL RIGHTS RESERVED

DETERMINISTIC SIMULATIONS OF SHARED MEMORY ON BOUNDED DEGREE NETWORKS

Kieran T. Herley, Ph.D.

Cornell University 1990

The Parallel Random Access Machine (PRAM) is an abstract parallel machine conisting of a synchronous collection of n processors connected to a shared memory of m cells. The essential feature of the PRAM is that the processors can access any n-tuple of distinct cells in a single machine cycle. While the PRAM is an attractive and widely used framework for the design and analysis of parallel algorithms, it does not reflect the constraints of realistic multiprocessors. This thesis explores the problem of efficient deterministic simulations of PRAM computations on bounded degree networks of processors, a model of parallel machines closer to what can be built in practice.

It is shown that an arbitrary step of a PRAM with n processors and $m \geq n$ cells of shared memory can be simulated in

$$O\left(\log(m/n)\log n/\log\log n + \log n\log\log n(\log\log(m/n) - \log\log\log n)\right)$$

time in the worst-case on an n-node bounded degree network with a particular expander-based structure.

This simulation is more efficient than all deterministic simulations previously known both with respect to time and space. In the case where m/n is polylogarithmic in n, the worst-case time to simulate a single PRAM step is at most $O(\log n \log\log n)$ which is within a factor of $O(\log\log n)$ the diameter of the network. The space requirements for our algorithm are at most $O(m(\log(m/n))^3)$ overall. The simulation may also be adapted to run on to an n-processor augmented mesh-of-trees architecture with a running time of $O(\log n \log\log n(\log\log(m/n) - \log\log\log n) + \log(m/n))$

Overall these results suggest that, in principle at least, it is feasible to provide the abstraction of a shared memory on distributed models of parallel computation with only modest degradation in performance in the worst case.

Biographical Sketch

Kieran T. Herley graduated with a B.Sc. from University College Cork of the National University of Ireland in November 1982. He received an M.Sc. degree from the same institution in the following year. He enrolled in the graduate program at Cornell University in the autumn of 1983 and received the degree of M.S. in 1986.

To my parents.

Acknowledgements

This thesis could not have been completed without the generous help and constant encouragement of my advisor Gianfranco Bilardi. It has been a privilege to work with him.

I would also like to thank the other members of my committee, Sam Toueg and Peter Kahn, for their help and the faculty of the Computer Science department especially Charlie van Loan, and Abha Moitra for their support and encouragement.

I gratefully acknowledge the financial support of NSF grant DCR-86-02307, JSEP contract F49620-87-C-0044, and IBM agreement 12060043.

Finally, a sincere thank you to all those friends, housemates, officemates, ski-buddies, and others who have made my years here such an enjoyable and rewarding experience.

Table of Contents

Chapter 1

Introduction

The Parallel Random Access Machine (PRAM) is an abstract parallel machine that plays a central role in the study of parallel computation. A PRAM is essentially a set of synchronous processors connected to a common shared memory. The basic feature of the PRAM is that references to distinct memory cells can be made simultaneously by different processors. The PRAM represents an attractive framework for the development of parallel algorithms, a fact which is apparent from the abundance of PRAM algorithms that have been devised. (See [KR88] for a sample.) It is well known that is not feasible to build a parallel machine that allows processors unrestricted access to a common memory in unit time. This fact raises the question of whether it is possible to devise more feasible parallel machines, and algorithms capable of simulating PRAMs efficiently on such machines. This question has received a lot of attention in the literature and forms the subject of this thesis.

Two important constraints not captured by the PRAM model, but shared by most realistic multiprocessors, are the following

1. *memory granularity constraint:* the memory of the system is *segmented* into modules from which only one datum may be accessed per unit time.

2. *bounded degree constraint:* the components of the machine (processors and

memory modules *etc.*) are connected so that each is physically adjacent to only a small number of other components, the number being independent of the machine size.

The Bounded Degree Network (BDN) model is one that captures both of the above constraints. It consists of a synchronous collection of RAM [AHU74] processors each equipped with a private local memory. The processors are connected to one another by means of bidirectional channels. Examples of machines in this class include the cube-connected cycles [PV81], shuffle-exchange [Sto71], and mesh-connected machines, to name but a few.

With respect to the PRAM the BDN provides a more realistic model of parallel computers but is more restrictive from the perspective of the algorithm designer. In the PRAM model the n processors can access any set of n distinct memory cells, in the BDN model they are restricted to accessing cells stored locally. In addition the shared memory of the PRAM provides a more flexible communication medium than the message passing system of the BDN. In the case of the BDN, the algorithm designer assumes the additional burden of explicitly managing the distribution of the data among the local memories of the individual processors and choreographing its movement to ensure that each processor has the operands it requires at each point in the computation. In a sense the PRAM encourages the algorithm designer to focus on computational aspects of a parallel algorithm by providing him with an abstraction that hides the communication issues.

A tacit assumption underlying much of the work on PRAM algorithms is that a "good" PRAM algorithm for a problem captures enough of the parallel structure of the problem to make the algorithm suited to other parallel models such as the BDN. In other words, despite the fact that, as a *machine model*, the PRAM does not reflect the constraints of realistic multiprocessors, PRAM *algorithms* adapt well to other parallel machines. The hope is that in general a PRAM algorithm can be ported to a more realistic BDN M and still compare favourably with algorithms tailored specifically for M, in the sense that the advantages gained from working with the more convenient PRAM model outweigh any performance penalty that might be incurred. This philosophy underlies much of the literature on parallel algorithms.

A simulation scheme may be represented by a triple $(\mathcal{M}, \mathcal{O}, \mathcal{A})$, where $\mathcal{M}$ represents the structure of the underlying BDN, $\mathcal{O}$ (*memory organization*) represents the means by which the state of the PRAM memory is represented in the local memories of the individual BDN nodes, and $\mathcal{A}$ (*memory access algorithm*) represents the algorithm by which the state of the simulated memory is probed or updated.

A simulation scheme is said to have *slowdown* $\mathcal{S}$ if any arbitrary sequence of T PRAM steps can be simulated in

$$\mathcal{S} \cdot T$$

steps on the machine $\mathcal{M}$. Existing work has shown that bounded degree networks exist with simulation algorithms that have slowdown polylogarithmic in the size of the PRAM being simulated. This thesis is devoted to more precise investigation of the impact of the various machine parameters on the efficiency of simulations. The simulations schemes presented here are more efficient than all existing simulations, with respect to the resources of both time and space.

The obstacles to providing the abstraction of a shared memory on a distributed BDN model are nontrivial. Four main problems present themselves.

1. *communication latency:* Communication between processors can not be instantaneous since messages must be passed from processor to processor on their way from their origins to their respective destinations.

2. *conflicts:* Several variants of the basic PRAM model allow processors to access the same cell simultaneously. A mechanism has to be developed

to implement this feature on the BDN model.

3. *memory congestion:* Only one datum may be accessed from a memory module in unit time. Any simulation scheme that requires a large amount of information to be accessed from any individual BDN node is doomed to be inefficient.

4. *interprocessor coordination:* Coping with the above difficulties requires considerable coordination among the processors which must be achieved without excessive communication between processors.

1.1 Preliminaries

Throughout this thesis the quantities m, and n will have the following interpretation:

m the number of PRAM memory cells, and
n the number of PRAM processors.

It will generally be assumed that $m \geq n$. The cells of the shared memory of the PRAM will be referred to as *variables* to differentiate them from the cells in the local memories of the simulating machine.

Shared-memory multiprocessors have been mentioned in the literature at least as early as the mid sixties (see [Sch66]). Synchronous parallel shared memory machines in the spirit of the PRAM underly a number of early parallel algorithms (see [Csa76,Hir76,Pre78,Hir78] for example) although none of these studies gives a very precise formulation of the model. The PRAM model was formulated independently by Fortune and Wyllie [FW78] and Goldschlager [Gol78]. Variations on the basic PRAM theme can be found in many studies [Sch80,LPV81,SV81,BH85].

For the purposes of this thesis a PRAM will be regarded as a collection $P_1^{PRAM}, \cdots, P_n^{PRAM}$ of processors attached to a shared memory of m cells. The set of variables will be denoted by $U = \{u_1, \ldots, u_m\}$. The processors act synchronously in lock-step. During a single machine cycle processors may either access a single cell of memory or perform a single operation locally. Since simulation of arithmetic and logical operations is straightforward, we can regard a generic PRAM step simply as an n-tuple of memory accesses. A memory access can either be of the form *read*(u), $(u \in U)$, which results in the contents of the memory cell u being copied into a distinguished register of the executing processor, or of the form *write*(u,x), $(u \in U, x$ an integer value), which results in the integer value x being written to memory cell u. Simultaneous accesses to the same cell are allowed in varying degrees by different variants of the PRAM that have been defined in the literature [FW78,Sch80,SV81,LPV81, Gol78,BH85]. An Exclusive-Read Exclusive-Write (EREW) PRAM forbids simultaneous accesses entirely; a Concurrent-Read Exclusive-Write (CREW) PRAM permits several processor to read the same cell but forbids simultaneous writes to the same cell, and the Concurrent-Read Concurrent-Write (CRCW) PRAM allows simultaneous writes to the same cell subject to restriction that no processor may write to a cell that another is reading during the same cycle. For concreteness it is assumed here that the PRAM is of the CRCW variety, and that that when several processors attempt to write to the same memory cell one of them (chosen at random) succeeds. (Any other reasonable conflict resolution rule could also be adopted without altering the results.) For notational convenience a PRAM with n processors and m cells of shared memory will be referred to as an (n, m)-PRAM.

A BDN is a synchronous machine consisting of a p nodes (P_i, M_i), for $1 \leq i \leq p$, where P_i, and M_i represent a processor and its attached private random access memory respectively. These private local memories contain some number m' locations each. Each node is connected to a small number of other nodes by means of channels. The topology of the interconnection is specified as part of the design of the BDN. During a single machine-cycle, a BDN processor may perform a local operation, or communicate the contents of a single memory cell to one of its neighbours. It is assumed that each processor is equivalent in computational power to a single PRAM processor. The simulations described in this thesis assume that the BDN processor is able to perform the customary arithmetic and logical operations on small integer values ($O(\log m)$ bit arithmetic suffices). In general it will be assumed that p, the number of nodes in the simulating machine, and n, the number of PRAM processors, are equal. Unless stated otherwise n will denote their common value. Section 5.6 will briefly consider the case where $p \geq n$.

In this thesis we wish to explore the issue of simulations of PRAM algorithms on a model of computation that captures communication as well as computation costs. While it can certainly be argued that the BDN does not accurately model the cost of communication in realistic multiprocessors, it does provide a simple framework in which to investigate these issues.

In addition to being deterministic, the simulations considered in this thesis are *on-line*: the PRAM steps are simulated one at a time, the simulation of each step is completely independent of other steps.

In any simulation scheme, upon termination of the simulation of a given PRAM step, the state of the PRAM memory must be uniquely reconstructible from the state of the BDN memory. A simulation scheme is consistent if after the simulation of an arbitrary sequence of PRAM steps, the simulation of a step where processor P_i^{PRAM} executes *read*(u) results in loading the distinguished register of P_i with the correct value of u, which is of course the value assigned to u by the latest instruction of the form *write*(u,x).

1.2 Related Work

This section summarizes the literature on PRAM simulations on bounded degree networks.

Communication is a fundamental issue in any computations on a BDN. In the case of PRAM simulations, processors must communicate with one another to probe and update the simulated shared memory, and to coordinate their actions. A form of communication that underlies several of the simulations discussed later is that of *partial permutation routing* which involves the routing of n packets on an n-node BDN subject to the constraint that no node is the source or destination of more than one packet. We say that a parallel machine M can sort in time $S_M(N)$ if there exists a deterministic algorithm that sorts N records with integer keys in increasing order of key value in $S_M(N)$ time running on M. In most of the parallel machines M known to date any instance of the partial permutation routing problem can be routed on a machine M in $O(S_M(N))$ time. (For good deterministic routing algorithms that are not based on sorting, see [Upf89,LM89].)

In some restricted cases memory congestion poses no obstacle to efficient simulations. One such case occurs when the size m of the memory is within a constant factor of the number of processors in the simulating machine. In this case it suffices to distribute $O(1)$ cells per node of the BDN M. Accessing individual cells involves application of partial permutation routing techniques, and requires only $O(S_M(m))$ time. This observation has been made by numerous authors [LPV81,BH85,Sch80]. Another case in which congestion can be avoided occurs when the pattern of memory accessed by a PRAM over successive steps is predictable. For example, in the well known FFT algorithm [CT65] the pattern of memory accesses is fixed and independent of the values of the input data. The work of Vishkin and Wigderson [VW83] implies that any such oblivious PRAM computation can be simulated with slowdown $O(S_M(n))$

on a machine M. It is also possible to distribute the entries of a matrix so that any row of column may be accessed without congestion [LV82].

More frequently, however, m the number of memory cells is large in relation to n the number of PRAM processors, and the pattern of memory accesses is unpredictable. In this case the problem becomes more complex requiring more advanced techniques.

One approach to simulating large shared memories, proposed in [MV84] and further developed in [Upf84,KU88,Ran87,LPP88,Sie89], distributes the address space of the PRAM among the modules of the BDN according to some suitably chosen hash function. The objective is to ensure that, at any node of the network, memory congestion is low with high probability and hence obtain a good *expected-time* performance. In particular [Ran87] proposes a protocol for an n-node butterfly network that simulates $T \geq m/n$ steps of an n-processor PRAM with m memory cells in $O(T \log n)$ time (with high probability) assuming that m is polynomial in n. The protocol is deterministic, except for the random choice of the hash function that defines the address map.

Another approach, also proposed in [MV84], relies on representing the contents of a PRAM memory cell by using several BDN cells. This approach forms the basis for a number of deterministic simulation schemes [UW87,AHMP87, LPP87,HP89] including those proposed in this thesis. The goal of these schemes is to provide good *worst-case* performance.

In [UW87], a scheme is advanced to simulate an arbitrary step of an (n, m)-PRAM by an n-node BDN in time

$$\mathcal{S}_{UW}(m, n) \triangleq O(\log n \log m \log \log m). \tag{1.1}$$

The nodes of the BDN are interconnected as in the graph (V, E_{AKSL}), Leighton's modification of the sorting network due to Ajtai, Komlós, and Szemerédi [AKS83,Lei85]. This time bound was subsequently improved to

$$\mathcal{S}_{AHMP}(m, n) \triangleq O(\log n \log m) \tag{1.2}$$

by Alt, Hagerup, Mehlhorn and Preparata [AHMP87] using the same network. In both of these schemes, each PRAM variable is replicated and the copies are distributed among the nodes of the BDN. (This idea of employing replication to enhance the accessibility of data had been used before in the field of distributed systems. See [Gif79,Tho79] for example.) If the number of copies per variable is $2c-1$, then in order to access a variable a processor must access a *quorum* of at least c copies. Copies are timestamped at the time of writing so that a reading processor may recognize the correct value from among the copies it reads. For a particular variable some copies may be more accessible than others depending on how many processors are competing for access to each individual node. Processors have complete freedom in choosing which c copies to access and this flexibility allows them to coordinate their individual choices to minimize memory congestion. The distribution of the copies of the variables is modelled by a bipartite graph with a certain expander-like property that guarantees that for any set of at most n variables some quorum of copies is fairly accessible. The simulations considered in this thesis will also adopt this read/write discipline.

An

$$\Omega(\min\{(\log n \log m)/\log\log m), \sqrt{n \log n}\}) \tag{1.3}$$

lower bound for the time to simulate a step of a (n, m)-PRAM on an n-node BDN was established independently in [AHMP87] and [KU88]. The lower bound assumes that $m = \Omega(n^{2+\epsilon})$ for some $\epsilon > 0$ and also makes the so-called point-to-point assumption, that if different copies of a variable are read or updated then a separate message is dispatched to read or update each copy; these messages are transmitted by the network as distinct entities. (This lower bound will be refined in Chapter 8.) Hagerup, Mehlhorn, and Parker [HMP89] have established a lower bound of $\Omega(\log m)$ that does not assume that communication is point-to-point.

Notice the gap between the upper bound of (1.2) and the lower bound (1.3). In their paper, Alt *et al.* speculated that this disparity is due to the inadequacy of the lower bound. This is refuted by Theorem 1 below.

In [LPP87] a PRAM simulation is presented on a BDN consisting of an $n \times n$ mesh-of-trees [Lei81,NMB83], where the roots of the trees are RAM processors, while the other $\Theta(n^2)$ nodes are packet-switching elements. The regularity and simplicity of the interconnection make it attractive for VLSI implementations. For m polynomial in n, a PRAM step is simulated in time

$$\mathcal{S}_{LPP}(m,n) \triangleq O(\log^2 n / \log\log n). \tag{1.4}$$

Hornick and Preparata [HP89] have argued that in any VLSI realization of a (sufficiently large) shared memory, the layout area is likely to be dominated by the area for the local memories of the individual nodes, rather than the area required for the interconnection network. They argue that, at least in the context of VLSI realizations, p the number of nodes of the simulating machine can be allowed to grow as a function of m, without affecting the layout area. In [HP89] they present a simulation algorithm with the same running time as [LPP87] using $O(1)$ copies per variable on a $n^{1+\epsilon} \times n^{1+\epsilon}$ mesh-of-trees, for some $\epsilon > 0$ again under the assumption that m is polynomial in n.

All of the literature cited above deals with the simulation of PRAMs of the EREW variety. This restriction is not essential. Vishkin [Vis83] has shown that the problem of simulating a CRCW PRAM step can be reduced to the problem of simulating a related EREW PRAM step at a cost of of adding an $O(S_M(n))$ factor to the running time. As a consequence of this result a CRCW PRAM can be simulated as efficiently as the more restrictive EREW PRAM on any reasonable bounded degree network. In this sense the two PRAM variants are equivalent.

1.3 Summary of Results

Despite the many studies devoted to the question of efficient PRAM simulations, some of the issues are not yet well understood. Of particular concern, is the question of how each of the parameters m, p, and n influence the slowdown of the simulations.

For example, the lower bound (1.3) quoted earlier suggests that the simulation time should be sensitive to the ratio m/p of the size of the shared memory to the number of processors in the simulating machine. From a deeper understanding of the importance of this quantity we obtain a simulation much faster than any previously known in the case when m/n is small. The class $\mathcal{NC}$, as it relates to the PRAM, represents the class of problems for which there is a deterministic PRAM algorithm that employs $n^{O(1)}$ processors and runs in time $(\log n)^{O(1)}$, where n is the problem size. The class is often regarded as the set of problems amenable to efficient parallel solutions. Many such algorithms use a shared memory of size m larger than the number of processors available p by at most a polylogarithmic factor. Thus, simulations of small shared memory is especially interesting. Also unknown is the extent to which extra hardware resources in the form of additional processors or memory modules can be exploited to speed up simulations. The main result of this thesis illuminates how these parameters constrain the slowdown that can be achieved.

The central result of this thesis is as follows.

Theorem 1 *For every m, and n with $n \leq m \leq n2^{2^{\sqrt{\log n}}}$ there is a n-node BDN, and a deterministic algorithm that can simulate any step of an (n,m)-PRAM in*

$$\begin{aligned} \mathcal{S}_{BDN}(m,n) &= O(\log(m/n)\log n/\log\log n \\ &\quad + \log n \log\log n(\log\log(m/n) - \log\log\log n)) \end{aligned}$$

time. This simulation requires

$$O((m/n)(\log(m/n)/\log\log n)^2(\log\log(m/n) - \log\log\log n))$$

storage per node.

(In fact we will see that faster simulations than these are possible in the case where p, the number of nodes in the simulating machine. is greater than n.) It can be seen that in general

$$\mathcal{S}_{BDN}(m,n) = o(\mathcal{S}_{AHMP}(m,n)).$$

The improvement is especially marked when the ratio m/n is small. As discussed earlier many efficient algorithms fall in this class. For example, if $m = n(\log n)^{O(1)}$ then the time bound simplifies to $O(\log n \log\log n)$. This represents an improvement of a factor of $O(\log n/(\log\log n)^2)$ over the simulation of [AHMP87]. This upper bound of $O(\log n \log\log n)$ is within a factor of $O(\log\log n)$ of the both the lower bound imposed by the diameter of the BDN and the expected running time of the simulation algorithm of [Ran87].

The deterministic simulation of the last section as well as those considered here have the properties that (i) the locations of the copies of the individual variables do not vary with time, (ii) the simulations adopt the majority read/write discipline discussed earlier. (iii) the simulations are point-to-point as defined in the previous section, and (iv) the simulations are on-line *i.e.* the simulation of each step is independent of all other steps. In Chapter 8 an informal argument is presented that suggests that any simulation obeying the above four conditions must have slowdown of

$$\Omega\left(\min\left[\frac{\log n \log(m/n)}{\log\log n}, \sqrt{n}\log p\right]\right).$$

This suggests that any simulation that is faster than that of Theorem 1 will have to violate one of these conditions. No existing simulation exploits this freedom in any essential way. Chapter 8 also includes a formal proof of the following lower bound.

Theorem 2 *If $T \geq (1+\epsilon)(m/n)$, and $m \geq p \geq n$ for some constant $\epsilon > 0$, then the worst-case simulation time for a straight-line SIMD program running for T steps of an (n,m)-PRAM on a p-node BDN is*

$$\Omega\left(T(n/p)\min\left[\frac{\log^2(m/n)}{\log\log(m/n)}, \sqrt{p\log(m/2p)}\right]\right).$$

A similar result has been obtained independently by Hagerup, Mehlhorn, and Parker [HMP89].

All of the previous deterministic studies [UW87,AHMP87,LPP87] have one major drawback. They all require that each node in the simulating BDN maintain a table of size m in its local memory to encode the memory organization. The simulations of [AHMP87] for instance, require only $O((m/n)d)$ storage per node (where d is the number of copies per variable) to store copies of values assigned to that node, but $O(md)$ storage per node to encode the memory organization. The total amount of memory required is $O(dmn)$, which is greater than the size of the memory being simulated by a factor of dn in contrast with a factor of $O((\log(m/n)/\log\log n)^2(\log\log(m/n) - \log\log\log n))$ in the case of Theorem 1. Space-efficient simulations are especially important in the context of VLSI realizations of PRAMs where the internal complexity of the nodes must be considered.

The implementation of the simulation of Theorem 1 relies on a handful of routing primitives such as sorting, parallel prefix [KRS85], and a certain form of routing. The expander-based BDN underlying this simulation is chosen to support these operations optimally. Any other network that supports these operations efficiently will also provide a platform for efficient PRAM simulations although not necessarily with the same efficiency as that of Theorem 1. This point is illustrated by the following result which shows that efficient simulations are possible on mesh-of-trees architecture augmented so that the processors are joined in the structure of a complete binary tree. Let this augmented mesh-of-trees architecture be denoted by AMOT.

Theorem 3 *For every m, and n with $n \geq m \leq n2^{\sqrt{n/2}+1}$ there is a deterministic algorithm that can simulate any step of an (n, m)-PRAM in*

$$\mathcal{S}_{AMOT}(m, n) \triangleq O((\log(m/n) + \log n \log\log n)(\log\log(m/n) - \log\log\log n))$$

time using

$$O((m/n)(\log(m/n))^2(\log\log(m/n) - \log\log\log n))$$

storage per node on an $n \times n$ AMOT.

This is faster than the running time $\mathcal{S}_{LPP}(m, n)$ of the simulations of of Luccio *et al.* [LPP87] and those of Hornick and Preparata [HP89] in most cases. In fact, Chapter 6 presents an algorithm with a running time of

$$O(\log(m/n) + \log\log\log n(\log\log(m/n) - \log\log\log n))$$

which is faster than $\mathcal{S}_{AMOT}(m, n)$ in some cases, but this algorithm does not meet the stipulated space bounds.

The simulation on the mesh-of-trees is faster than the simulation on a BDN of the type specified in Theorem 1 with the same number of processors. Indeed the upper bound of $\mathcal{S}_{AMOT}(m, n)$ on an $n \times n$ augmented mesh-of-trees is less than the lower bound (1.3) established for simulations on any n-node BDN. This apparent anomaly may be explained as follows. The lower bound relies on the fact that an n-node BDN has only n communication channels and that the processors participate in both computation and communication. In the $n \times n$ MOT model the row and column trees provide the communication medium with $\Theta(n^2)$ channels. All of the work in message passing is performed by the channels and switches of these trees. The processors themselves play no part in communication other than transmitting and receiving packets. Thus the lower bound does not apply to the mesh-of-trees. The row and column trees with $\Theta(n^2)$ communication channels provide a more powerful communication medium then the n channels of the BDN of Theorem 1. This extra power is exploited to obtain a faster simulation.

In order to put these results in perspective, the paragraphs that follow discuss some of the difficulties in implementing fast simulation schemes and suggest some the ideas used to solve them. For the purposes of the discussion that follows it is assumed that the number of processors in the simulated PRAM and the simulating BDN are equal.

The previous studies [UW87,AHMP87,LPP87,HP89] all share the following structure. The variables are accessed in *stages*; each stage succeeds in accessing a quorum of copies for all but a small fraction of the outstanding variables. Each stage involves a global communication step and so requires $\Omega(\log n)$ time owing to the latency of the network. The number of outstanding variables decays rapidly with each passing stage, and so the number of variables accessed per unit time decreases from each stage to the next. One of the shortcoming of these schemes is that they fail to exploit the communication capabilities of the underlying BDN fully.

In contrast to the structure of the previous simulations, the simulation presented later in this thesis has the following two part structure

1. Decide which copies to access, and

2. Access the selected copies.

The two functions of deciding which copies to access, and physically accessing them were combined in the previous studies. (The decision making was implicit in the guarantee that a fraction of the variables died during each stage.) Divorcing these two tasks allows the processors greater flexibility in coordinating their individual choices to minimize congestion.

Memory organizations of the type considered here, can be modelled conveniently as a bipartite graph $G = (U, V : E)$ where U is the set of PRAM variables, V is the set of the nodes of the BDN, and an edge $(u, \gamma(u))$ represents the fact that the node $\gamma(u)$ contains a copy of the variable u.

In this framework, the collective decision among the processors concerning which copies of a set Z of variables to access is essentially one of constructing a subgraph H_Z of G such that (i) the maximum degree (*congestion*) of any vertex is modest, and (ii) H_Z contains a quorum of edges for each u in Z.

Given H_Z the problem of physically accessing the corresponding copies is a routing problem of the type discussed below.

To achieve the best possible performance during the second part of the simulation, it is necessary to make sure that the processors and the communication channels are fully exploited. This suggests that the work should be distributed among the various nodes and channels as evenly as possible. This motivates the need for a memory organization that bounds the amount of information that has to be accessed from any individual node, as well as a network and efficient routing algorithm to distribute the routing workload as evenly as possible among the nodes and channels of the network.

The problem of physically accessing the copies corresponding to some subgraph H_Z is an instance of the following routing problem. The (N, K_1, K_2)-routing problem involves routing N packets on a BDN subject to the constraint that no node is the source of more than K_1 packets or the destination of more than K_2 packets. This problem was introduced in [PU89] and a solution was presented there that is optimal with respect to the number of communication steps performed. By modifying the techniques of [PU89], an algorithm that routes any instance of the (N, K_1, K_2)-routing problem is advanced in Chapter 2 that runs in $O(\lceil N/n \rceil \log p + K_1 + K_2)$ time on a n-node BDN with a particular expander-based structure. An algorithm for the same problem with running time

$$O(\lceil N/n \rceil + \log p + K_1 + K_2)$$

on a $n \times n$ MOT is presented in Chapter 6.

We say that a set $S \subseteq V$ of nodes *confines* a variable u if a quorum of the copies of u lie in S. Let $\mathcal{C}(S)$ denote the set of variables confined in S. Accessing the set $\mathcal{C}(S)$ of variables involves accessing at least one copy per variable from the nodes in S, and so some node in S must suffer from congestion at least $\Omega(|\mathcal{C}(S)|/|S|)$. Let

$$q_G \triangleq \max_{S \subseteq V} \left(\frac{\min(\mathcal{C}(S), n)}{|S|} \right). \tag{1.5}$$

This is a lower bound on the worst-case time to access n variables. It turns out that the size of the smallest set of modules containing a given number of variables depends strongly on the value of d the number of copies per variable: the smaller this value the smaller the size of the set of modules. This suggests that the best q_G achievable decreases as d increases. A careful analysis reveals a tradeoff on the number of copies. Roughly speaking, decreasing d reduces the number of copies to be processed, but creates higher congestion at memory modules, thereby making the job of accessing those copies more difficult. Introducing the notion of a generalized expander we investigate the trade-off between the number of copies per variable and memory congestion. The choice of an appropriate point in this trade-off is critical in order to balance the work needed to extract the relevant information from the memory modules with the work needed to route this information to the intended processors.

The task of constructing H_Z is non-trivial. The time bounds stipulated in Theorem 1 are quite restrictive, especially for small values of m/n. The decision of which copies (edges) of each variable to include in H_Z requires considerable coordination among the processors. The goal is to achieve this coordination but avoid expensive global communication as much as possible, by tailoring the algorithm to ensure that most of the interaction is between adjacent processors. The key idea is to devote teams of processors. corresponding to physical neighbourhoods of the BDN. to construct fragments of H_Z corresponding to neighbourhoods of that graph. Communication within a team will always be between physically adjacent nodes and hence cheap. Communication between teams on the other hand is expensive and is avoided by having each team recompute intermediate results rather attempt to communicate with another team. The algorithm trades redundant computation for communication.

The problem of achieving the stated space-bounds may be abstracted as follows. Represent an m-tuple of values $A = (a_1, \ldots, a_m)$ in the local memories of the nodes on the BDN. Clearly there a tradeoff between the space requirements and accessibility: simply replicating A at each node renders it very accessible but is extravagantly wasteful of space; on the other hand distributing the values $a_1, \ldots, a_m$ among the nodes m/n per node is space efficient but not very accessible. The question is whether there is a happy medium that is efficient both with respect to time and space. This problem does not appear to have been investigated previously. Chapter 7 provides an affirmative answer to the above question.

1.4 Guide to the Remainder of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces some routing algorithms that underly the simulations of later chapters. Chapter 3 is devoted to the development of certain graph structures called generalized expanders that capture the notion of a "good" memory organization. All of the memory organizations of the simulations in this thesis embody graphs of this type. Chapter 4 presents a restricted case of Theorem 1. Although this result is subsumed by Theorem 1, it is of interest because a far less complex algorithm suffices in this case. In Chapter 5 a weakened version of the theorem is presented, that is identical to Theorem 1 except that it does not meet the stated space bounds at each node. Chapter 6 shows how to modify the algorithm to run on the augmented mesh-of-trees. Chapter 7 improves the space bounds of the simulations presented in Chapters 5 and 6. A lower bound for the problem is presented in Chapter 8. Finally, Chapter 9 offers some concluding remarks. Portions of this work have appeared in [HB88,Her89a,Her89b].

Chapter 2

Operations on Distributed Sets

Often the need arises to process sets of items distributed among the nodes of a BDN. Examples of such manipulations include the operations of sorting, parallel prefix, packet routing, load balancing to name but a few. In this chapter, a number of such algorithms are developed that will prove useful later on.

2.1 Distributed Sets

A *distributed set* on a network $G = (V, E)$ is a pair $\hat{Z} \triangleq (Z, f)$ where Z is a set of packets and $f : Z \to V$ specifies that packet $x \in Z$ is at node $f(x)$. The *profile* $\pi(\hat{Z})$ of $\hat{Z}$ is an p-tuple $(k_1, k_2, \ldots, k_p)$, where $p = |V|$ and k_i represents the number of packets of Z mapped to node i by f. The *degree* $\Delta(\hat{Z})$ of $\hat{Z}$ is the maximum number of packets assigned by f to any one node.

One of the most fundamental operations in parallel computation is that of packet routing. An instance of a routing problem may be characterized by a pair of distributions $\phi \triangleq (f_1, f_2)$. An algorithm is said to route ϕ if it transforms the distributed set (Z, f_1) into (Z, f_2), *i.e.* it can move every packet x from its *source* $f_1(x)$ to its *destination* $f_2(x)$.

The pair (f_1, f_2) represents an instance of the *partial permutation routing* *problem* if both f_1 and f_2 are one-to-one functions. Any instance of the partial permutation routing problem can be routed in $O(\log p)$ time on the BDN (V, E_{AKSL}) with the structure of Leighton's modification of the p-node sorting network of [AKS83,Lei85].

More generally, the (N, K_1, K_2)-*routing problem* involves routing N packets, subject to the constraint that at most K_1 packets have the same source and at most K_2 have the same destination. This problem was formulated by Peleg and Upfal [PU89]. Their network is essentially a *square root expander* [LPS86] augmented by the sorting network of [AKS83,Lei85]. It is shown that a distributed set Z of degree K can be balanced in $O(\lceil |Z|/p \rceil + K)$ time. However their results are quoted in terms of a model in which only communication steps are counted.

The next section describes a BDN and presents an optimal deterministic algorithm, based on the algorithm of Peleg and Upfal, that routes any instance of the (N, K_1, K_2)-routing problem in

$$O(\lceil N/p \rceil \log p + K_1 + K_2)$$

time, counting both local computation steps and communication steps.

The routing algorithm mentioned above depends on the idea of *balancing.* (Balancing is referred to as *token distribution* in [PU89].) The set $\hat{Z}$ is *balanced* if it has degree $\lceil |Z|/p \rceil$, and approximately balanced if it has degree $O(\lceil |Z|/p \rceil)$. A distributed set $\hat{Z} = (Z, f)$ of packets labelled with integer values is said to be *balancesorted* if it is balanced and the packets with the $\lceil |Z|/p \rceil$ smallest keys lie in node 1, the next $\lceil |Z|/p \rceil$ next smallest lie in node 2 and so on. The next section describes an optimal balancesorting algorithm.

A *distributed dictionary D* is a distributed set of packets each labelled with a unique *key*, and some *value.* A packet may be accessed by specifying its key, rather than its location. A simple implementation of a parallel distributed dictionary on a BDN is given in the last section of this chapter.

2.2 Generalized Packet Routing

This section studies the problems of balancing and routing on bounded degree networks.

The balancing of [PU89] is achieved in a sequence of *diffusions*, each of which reduces the degree of the distribution by some fixed factor $\gamma < 1$. Associated with each diffusion is a directed subnetwork of the BDN called a *flowdag*. During a diffusion processors transmit packets only along the edges of the corresponding flowdag. In some cases the overhead in constructing these flowdags dominates the work required to move the packets. The idea is to decompose the initial problem of balancing p packets on an p-node BDN, to some number s essentially identical problems of balancing p/s packets each on a subnetwork of size p/s. It is easier to construct the flowdags for the smaller problem within the stipulated time bounds.

In the next subsection the algorithm of [PU89] is sketched. In the subsequent subsection the modifications necessary to achieve the stated time bounds are described.

2.2.1 The Method of Peleg and Upfal

This section paraphrases the results of [PU89].

Let $G = (V, E)$ represent the structure of a bounded degree network. A directed acyclic subgraph $D = (V, E_D)$ of the graph G is a *flowdag* for a subset $U \subseteq V$ if, for some l depending only on G, (i) each $v \in V$ has indegree at most l in D, and (ii) each $u \in U$ has outdegree at least $l + 1$ in D.

Lemma 1 (**[PU89]**) *There is a BDN (V, E_{bal}) such that for every $U \subseteq V$ of size at most $\beta|V|$ (β a constant, independent of $|V|$), there is a flowdag D_U. Moreover there is a deterministic algorithm that allows each node in U to determine its neighbours in D_U in $O(\log |U|)$ time.*

Let $\hat{S} = (S, f)$ be a distributed set with profile $\pi(\hat{S}) = \mathbf{k} = (k_1, k_2, \ldots, k_p)$. Define $U(\hat{S}) \triangleq \{v_i \in V | k_i \geq (1/2)\Delta(\hat{S})\}$ and let $D(\hat{S})$ represent the flowdag $D_{U(\hat{S})}$. Let *Flow*$(\hat{S}, t)$ denote the operation of moving t packets from x to y for each directed edge (x, y) in $D(\hat{S})$.

Lemma 2 (**[PU89]**) *There exist constants $\gamma < 1$ and $\rho < 1$ such that for all distributed sets $\hat{S}$ of size at most p, the degree of the distributed set after the execution of* Flow$(\hat{S}, \rho\Delta(\hat{S}))$ *is at most $\gamma\Delta(\hat{S})$.*

Let an application of *Flow*$(\hat{S}, \rho\Delta(\hat{S}))$ be referred to as a *diffusion* step. The above lemma suggests the following straightforward balancing algorithm consisting of a sequence of *stages*. The i^{th} stage is applied to the distributed set $\hat{S^{(i)}}$ resulting from the first $i-1$ stages. During each stage a flowdag $D_i = D(\hat{S^{(i)}})$ is built, and then a diffusion *Flow*$(\hat{S^{(i)}}, \rho\Delta(\hat{S^{(i)}}))$ is performed. The i^{th} diffusion runs in $O(\gamma^i\Delta(\hat{S}))$ steps, but the corresponding flowdag requires $O(\log n)$ time to build, so the algorithm spends $O(\sum \gamma^i\Delta(\hat{S})) = O(\Delta(\hat{S}))$ time actually moving packets during flowphases, but spends $O(\log \Delta(\hat{S}) \log n)$ time building flowdags. When balancing distributed sets of small degree the overhead due to flowdag construction dominates and so this algorithm does not meet the desired $O(\lceil |S|/p \rceil \log p + \Delta(S))$ time bound in all cases.

The next section shows how this overhead can be reduced so that it is always subsumed by the work required for the movement of packets. A key element is the following lemma due to Peleg and Upfal that shows that it is possible to construct a sequence of subsets $U'_1, U'_2, \ldots$ efficiently that approximates the sequence $U(\hat{S^{(1)}}), U(\hat{S^{(2)}}), \ldots$ in the sense that the corresponding set of flowdags $D'_1, D'_2, \ldots$ can be used in place of $D_1, D_2, \ldots$ during balancing.

Lemma 3 (**[PU89]**) *for each $\hat{S} = (S, f)$ there is a sequence of subsets of V $U'_1, U'_2, \ldots, U'_H$, where (i) $U'_1 = U(\hat{S^{(1)}})$, (ii) $U(\hat{S^{(j)}}) \subseteq U'_j$ for each $1 \leq i \leq H$,* *(iii)* $\gamma^H \Delta(\hat{S}) = O(1)$, *(iv)* $|U_j'| \leq \beta|V|$ *for* $1 \leq i \leq H$, *and (v) each* U_{j+1}' *can be computed from* U_j' *in* $O(\log\log p)$ *time.*

2.2.2 An Optimal Balancing Algorithm

The BDN underlying the algorithm presented here has the structure

$$(V, E_{route}) \triangleq (V, E_{exp} \cup E_{AKSL} \cup E_{tree}).$$

The nodes of V are arranged in the pattern of a $N_1 \times N_2$ array where $N_1 \triangleq \log\log p$ and $N_2 \triangleq p/N_1$. All the rows of V has the same structure, that of a square root expander[LPS86] with N_2 nodes. Similarly all the columns have the structure of a square-root expander on N_1 nodes. All of the edges in the row and column expanders together constitute the set E_{exp}. The edges of E_{AKSL} give the nodes the structure of the p-node sorting network of [AKS83, Lei85] that can sort p integers in $O(\log p)$ time: those of E_{tree} are the edges of a complete binary tree with p nodes.

First a balancesorting algorithm is outlined, the solution of the more general (p, K_1, K_2)-routing problem will follow from the results of [PU89].

Let $\hat{S} = (S, f)$ be a be a distributed set of size at most p. (The balancing of larger sets will be described later on.) The overall structure of the algorithm is shown below.

BalanceSort

I Approximately balance the packets in each column.

II Approximately balance the packets in each row.

III Sort the packets.

Let $k_{i,j}$ represent the number of packets initially held by node (i, j), and let m_j denote the maximum number of packets held by any node in the j^{th} column following Step I. After column balancing, m_j can be at most $(A/N_1)\sum_{i=1}^{N_1} k_{i,j}$ for some constant A. Immediately prior to Step II no row can contain more than $\sum_{j=1}^{N_2} m_j = O(N_2)$ packets. Thus Step II involves balancing $O(N_2)$ packets in each row. The distribution $\mathbf{k}^{(i)} = (k_1^{(i)}, k_2^{(i)}, \ldots, k_{N_2}^{(i)})$ of packets in row i is *dominated* by $\mathbf{m} \triangleq (m_1, m_2, \ldots, m_{N_2})$, in the sense that $k_j^{(i)} \leq m_j$ for $1 \leq j \leq N_2$. In a sense Step I has decomposed the initial balancing problem into N_1 essentially identical subproblems.

Following the approximate balancing of each row during Step II each node holds at most $O(1)$ packets. A constant number of sorting steps suffices to complete the balancesorting in Step III.

The processors of the BDN can compute the value of $\Delta(\hat{S})$ in $O(\Delta(\hat{S}) + \log p)$ time. The straightforward balancing method mentioned in the previous section is used to perform Step I in $O(\log(\Delta(\hat{S}))\log N_1 + \Delta(\hat{S}))$ time. This quantity is bounded by $O(\Delta(\hat{S}) + \log p)$ since $N_1 = \log\log p$. Step III can be accomplished in $O(\log p)$ time, applying the sorting techniques of [AKS83, Lei85].

If $\Delta(\hat{S}) = \Omega(\log p \log\log p)$, then the same method as Step I is employed for Step II, otherwise the method outlined below is adopted. For the remainder of this section it will be assumed that $\Delta(\hat{S}) = o(\log p \log\log p)$.

The distribution $\mathbf{m}$ dominates the distribution of packets in each row after Step I. The key point is that same set of flowdags will suffice for all rows during Step II, and the overhead of constructing the flowdags may be shared among the rows.

Step II

(1) Compute m_j for each column j. Let $U_1' = U(\mathbf{m})$.

(2) Compute the sequence of sets $U_1', U_2', \ldots, U_H'$ in the first row.

(3) Distribute the U_j' evenly among the rows.

(4) For each U_j' assigned to row i compute the corresponding flowdag D_j' in row i.

(5) Let the processors within each column j pool adjacency information so that each processor in the column knows the neighbours of the j^{th} vertex in each of flowdags D'_1, $\ldots, D'_H$.

(6) Execute the flowphases in sequence. (Do this concurrently in each row.)

Each column has a bounded degree tree of $O(\log N_1)$ depth as a subgraph, since each column expander has diameter $O(\log N_1)$, and so Substep 1 can easily be executed in $O(\Delta(\hat{S})+\log N_1)$ time in each column using these column trees. Substep 2 involves the computation of sequence of H sets, each of which can be computed from the previous one in the sequence in $O(\log\log N_1)$ time by Lemma 3. Since $\Delta(\hat{S}) = o(\log p \log\log p)$ by assumption, H must be at most $O(\log\log p) = O(N_1)$. In other words at most $O(1)$ distributions are assigned to each row during Substep 3. Thus Substeps 3 and 5 can be accomplished in $O(\log N_1)$ time using the column trees. The construction of the flowdags in Substep 4 can be completed in $O(\log p)$ time by Lemma 1. By Lemma 2, Substep 6 can be completed in $O(\Delta(\hat{S}))$ time. Summing the contributions of all these substeps, and simplifying, the total running time for Step II is seen to be $O(\log p + \Delta(\hat{S}))$.

This establishes the result in the case where $|S| \leq p$. Sets of larger size can be handled by grouping packets into parcels of size $|S|/p$, approximately balancing the parcels (treating each parcel as a unit) with the method just described. Since there are $O(p)$ such parcels, and each flowstep requires $O(\lceil |S|/p \rceil)$ time, balancing can be accomplished in $O(\lceil |S|/p \rceil \log n + \Delta(\hat{S}))$ time. The individual packets can be sorted using the techniques of [AKS83,Lei85,BS78].

Lemma 4 *There exists a BDN* (V, E_{route}) *and a deterministic algorithm that balances any distributed set* $\hat{Z} = (Z, f)$ *in* $O(\lceil |Z|/p \rceil \log p + \Delta(\hat{Z}))$ *time.*

By incorporating this modified balancing algorithm into the routing algorithm of [PU89], the following result is achieved.

Theorem 4 *There is an p-node BDN* (V, E_{route}) *and a deterministic algorithm that routes any instance of the* (N, K_1, K_2)*-routing problem in* $O(\lceil N/p \rceil \log p + K_1 + K_2)$ *time.*

This is optimal according to a lower bound of [PU89].

2.3 A Simple Parallel Dictionary

A *parallel dictionary* is a data type consisting of a collection $\hat{D}$ of *records* each labelled with a unique identifying *key*, and a *value*. The value of a record with a particular key x may be probed by use of the *lookup*$(\hat{D}, x)$ operation. The value v may be written to a record with a key x by means of the *update*$(\hat{D}, x, v)$ operation. The processors may execute a p-tuple of *lookup* operations, or a p-tuple of *update* operations simultaneously. Concurrent accesses to the same record are permitted; in the case of concurrent updates one of the update (chosen arbitrarily) succeeds. Updates to non-existent records result in the creation of the appropriate record; lookups to non-existent records return a special value to indicate this fact.

PRAM implementations of parallel dictionaries exist based on hashing techniques [FKS84,DKM+88], and on parallel analogues of 2-3 trees [PVW88]. The algorithms of [Ran87] and [KU88] provide an implementation of a parallel dictionary with good expected performance on a BDN provided that the range of the keys is polynomially bounded in p.

In the paragraphs that follow a simple deterministic implementation of a parallel dictionary on a bounded degree network is presented which will prove a useful tool in the simulations of later chapters. Almost certainly, more efficient implementations are possible in the case where $|\hat{D}| \gg p$. However, since in all of our applications $|\hat{D}| = O(p)$, this will not be pursued.

The records of the dictionary are maintained as packets one per record in the distributed set $\hat{D}$. Each packet is labelled with the key and value of the corresponding record. The distributed set is maintained balanced and sorted. The BDN has the structure of (V, E_{route}) defined in the previous section.

A p-tuple of *lookups* may be executed as follows. (It is assumed for ease of presentation that no two processors attempt to lookup the same record simultaneously; this restriction can be removed by means of standard techniques as found in [Vis83].)

(i) For each request *lookup*$(\hat{D}, x)$ generate a request packet labelled with x.

(ii) Sort the set of request packets by key.

(iii) Merge the sorted set of request packets with the set of dictionary packets.

(iv) Pair each request packet with the dictionary packet bearing the same key (they are adjacent in the merged sequence), and load the appropriate value into the request packet.

(v) Rebalance the dictionary packets, and route the request packets back to their respective origins by retracing the path taken in the earlier steps.

The running time of this may be analysed as follows. Steps (i) and (iv) can be executed locally at each node in $O(1)$ time. Step (ii) can be accomplished using the sorting scheme of [AKS83,Lei85] in $O(\log p)$ time. Step (iii) involves merging two balanced set of size $|\hat{D}|$ and p packets and this is completed using the techniques of [AKS83,Lei85,BS78] in $O(\lceil |\hat{D}|/p \rceil \log p)$ time. The rebalancing of Step (iv) uses the balancing algorithm of the previous section and runs in $O(\lceil|\hat{D}|/p\rceil \log p)$ time. Summing the contribution of all the steps it may be seen that the above may be completed in $O(\lceil|\hat{D}|/p\rceil \log p)$ time. A p-tuples of *updates* may be executed within the same time bound.

Chapter 3

Generalized Expanders

This chapter introduces the notion of a generalized expander that forms the basis for the memory organizations in the chapters that follow. The existence of such graphs is demonstrated and certain properties of these graphs are derived that serve as the basis of the memory access algorithms in later chapters.

3.1 Replication and Consistency

As discussed in Chapter 1, any memory organization that maintains only one copy of each variable is doomed to suffer from poor performance due to memory congestion. Mehlhorn and Vishkin [MV84] proposed replicating each variable. The idea is that while some copies are more difficult to access than others due to memory congestion, by choosing the distribution carefully it is possible to ensure that at least *one* copy of each variable can be accessed with relative ease. The difficulty with this approach is that in order to update a variable, all of its copies need to be updated. Upfal and Wigderson[UW87] proposed a scheme where each variable is replicated d times and each reading or writing processor must access a *quorum* of at least $\lfloor d/2 \rfloor + 1$ copies. (Schemes of this type had been employed before by researchers in the area of distributed systems [Gif79, Tho79].) Upfal and Wigderson recognized that it is possible to distribute the copies in such a way that, for any set of n variables, a quorum of copies can be accessed without causing too much congestion at any individual node. This idea has been adopted in subsequent work [AHMP87,LPP87,HP89] including this thesis.

The following read/write discipline mentioned in Chapter 1 is adopted. A writing processor writes a quorum of copies, and timestamps each copy to reflect the current time. A reading processor reads a quorum of copies and selects the value with the most recent timestamp. Since each quorum has size $\lfloor d/2 \rfloor + 1$, each processor attempting to read a variable u is guaranteed to access at least one of the copies written by the most recent write operation to that variable. This value will be the one with the most recent timestamp.

It should be noted that in general it suffices to require each reading processor to read r copies of each variable and to require writing processors to write w copies as long as $w + r$ exceeds the total number of copies per variable. This observation has been exploited to lower the cost of reading with respect to writing in distributed systems, where reading is generally a more common activity than writing.

3.2 Generalized Expanders

Let $G = (U, V; E)$ be a bipartite graph, such that $|U| = m$, and $|V| = p$. The graph G may be interpreted as a memory organization as follows. The vertices of U represent the variables, the vertices in V represent the nodes of the network. The neighbours $\gamma_1(u), \gamma_2(u), \ldots, \gamma_d(u)$ of a vertex $u \in U$ represent the nodes in the network holding copies of the variable represented by u.

A set F of edges is called a *k-bundle* for $Z \subseteq U$ if (i) each edge in F is incident on some vertex in Z, and (ii) the degree of each $u \in Z$ is exactly k with respect to the edges of F. Let $\Gamma_F(Z)$ denote the set of vertices in V touched by the edges in F, and refer to it as a *k-shadow* of Z. The expression

$\Gamma_E(Z)$ will be abbreviated to $\Gamma(Z)$.

Corresponding to each $Z \subseteq U$ and every bundle of edges F for Z, is a subgraph $(Z, \Gamma_F(Z); F)$ of G. For H a subgraph of G, let $deg(x, H)$ denote the degree of x in H, and define the *congestion* of H (denoted by $\Delta(H)$) to be the maximum degree of the vertices of V in H *i.e.*

$$\Delta(H) \triangleq \max_{v \in V} deg(v, H).$$

Definition 1 *A subgraph H of G is a (k,q)-subgraph for $Z \subseteq U$ if it contains a k-bundle of edges for Z, and the congestion $\Delta(H)$ is at most q. The edges of a (k,q)-subgraph for Z constitute a (k,q)-*sheaf *for Z.*

Accessing the variables represented by Z involves accessing the copies corresponding to the edges of some $(\lfloor d/2 \rfloor + 1, q)$-sheaf F_Z for Z. Clearly accessing the copies represented by F_Z will require $\Omega(q)$ time. The minimum congestion q attainable and the size of the $(\lfloor d/2 \rfloor + 1)$-shadow $\Gamma_{F_Z}(Z)$ are related as follows:

$$\Delta(H_Z) \geq \frac{(\lfloor d/2 \rfloor + 1)|Z|}{|\Gamma_{F_Z}(Z)|}, \tag{3.1}$$

since $(\lfloor d/2 \rfloor + 1)|Z|$ copies are located in $|\Gamma_{F_Z}(Z)|$ nodes. Intuitively this suggests that a good memory organization should ensure that the copies of a set of variables are distributed over as many nodes as possible. These remarks motivate the following definitions.

Definition 2 *Let $G = (U, V; E)$ be a bipartite graph with $|U| = m$ and $|V| = p$, and with each node in U having degree d. Let c be an integer such that $1 \leq c \leq \lceil d/2 \rceil - 1$, and let $\sigma(k)$ be a real function of the integer k in the range $c \leq k \leq \lceil d/2 \rceil - 1$. Let $\lambda \leq 1$ be a positive real. We say that G is a $(m, p, \lambda, d, c, \sigma)$-generalized expander if, for every $S \subseteq U$ such that $|S| \leq \sigma(k)p$ and for every k-bundle F of S, where $c \leq k \leq \lceil d/2 \rceil - 1$,*

$$|\Gamma_F(S)| \geq \lambda k |S|. \tag{3.2}$$

Definition 3 *A graph* $G = (U, V; E)$ *is* balanced *if* $\Delta(G) = O(|E|/|V|)$.

For any $c \leq k \leq \lceil d/2 \rceil - 1$, subset $S \subseteq U$ of size at most $\sigma(k)p$, and k-bundle F for S, property (3.2) guarantees that the k-shadow of F has size at least $\lambda k|S|$. Considering that F contains only $k|S|$ edges in all, this implies that the average degree of the vertices in $\Gamma(S)$ with respect to the edges of F is $1/\lambda$. The expander property (3.2) holds for all k-bundles of all subsets $S \subseteq U$ upto a certain size determined by the threshold function σ.

A bipartite (a, b)-expander is a bipartite graph $G = (U, V; E)$ with $|U| = |V|$ such that each vertex has fixed degree (independent of $|U|$) and for all $S \subseteq U$ of size at most $a|U|$

$$|\Gamma(S)| \geq b|S|.$$

A $(n, n, \lambda, d, d, \sigma)$-generalized expander $G' = (U', V'; E')$ with $\lambda = b/d$, $\sigma = a$ is precisely the same as an (a, b)-expander of size n.

In the terminology of [DDPW83] a (m, p, l)-*concentrator* of depth one is a bipartite graph $G = (U, V; E)$ with $|U| = m$, $|V| = p$, such that for all subsets $Z \triangleq \{u_1, \ldots, u_{|Z|}\} \subseteq U$ of size at most l, there exists a set of independent edges $(u_1, v_1), \ldots, (u_{|Z|}, v_{|Z|})$. By Hall's Theorem (see [Har69]), this condition is equivalent to the assertion that for all $S \subseteq U$ of size at most l

$$|\Gamma(S)| \geq |S|.$$

An (m, p, l)-concentrator of depth one with degree d with respect to the vertices of U is a $(m, n, 1/d, d, d, l/n)$-generalized expander. Similarly any $(m, n, \lambda, d, c, \sigma)$-generalized expander with $\lambda \geq 1/c$ is an $(m, n, \sigma(d)n)$-concentrator.

The concentrator property guarantees expansion for d-bundles; the generalized expander has the additional property that this expansion property holds even for all k-bundles of Z, for $k \geq c$. The need for ensuring expansion for c-bundles may be understood as follows. The copies of a set of variables S are represented by the edges of a subgraph $H(S)$ of G. In order to access these variables the algorithms will refine $H(S)$ into a subgraph $H'(S)$ by discarding a carefully selected subset of the edges in $H(S)$ to ensure (i) that the congestion $\Delta(H'(S))$ is tolerable, and (ii) that $H'(S)$ contains a quorum of copies for each of the variables in S. This refinement takes place in some number r rounds each of which discards at most $d/2r$ edges belonging to each variable. In order to guarantee the correctness of this scheme it is necessary to ensure that the bundle of edges discarded at each round expands.

It is known that

$$\Theta\left(\frac{\log(m/l)}{\log(p/l)}\right) \tag{3.3}$$

edges per vertex are both necessary [NM82] and sufficient [DDPW83] for the existence of (m, p, l)-concentrators. No explicit construction of such concentrators with a number of edges matching the cited bounds are known in general; existence has been demonstrated by combinatorial argument. Various studies have given explicit constructions for special cases [GG81,Tan84].

Let $B_{m,p,d}$ be the set of all bipartite graphs of the from $G = (U, V; E)$. with labelled vertices and unlabelled edges, where $|U| = m$, $|V| = p$ and each $u \in U$ has degree d. The proof of the existence of generalized expanders is by combinatorial argument given by Lemmas 5 and 6 below. The argument is of the same type as that found in [UW87] for example. Essentially the idea is to place an upper bound smaller than $|B_{m,p,d}|$ on the number of graphs in $B_{m,p,d}$ that are not generalized expanders. This proof may be skipped without loss of continuity.

Lemma 5 *For all m,p, and d such that $m \geq p \geq 2d$ and for all $D \geq 4e(m/p)d$, at most $(p/2^D)|B_{m,p,d}|$ of the graphs in $B_{m,p,d}$ have a vertex of degree D or greater.*

Proof For every graph in $B_{m,p,d}$ with maximum degree greater than D, where $D \geq 4e(m/p)d$, there is a *witness* (S, v) such that (i) $S \subseteq U$ has size D, (ii) each $x \in S$ has an edge incident on $v \in V$. At most a fraction

$$\binom{p}{d}^{m-D}\binom{p}{d-1}^{D} \Big/ \binom{p}{d}^{m}$$

of the graphs in $B_{m,p,d}$ have given (S,v) as a witness. There are $\binom{m}{D}$ and p ways to choose S and v respectively. Making use of standard relations such as

$$\binom{a}{b} \le \left(\frac{ea}{b}\right)^{b}$$

$$\binom{p}{d-1} \Big/ \binom{p}{d} = \frac{d}{p-d+1}$$

f', the fraction of all graphs in $B_{m,p,d}$ with a vertex of degree d or more, may be bounded as follows

$$f' \le p\binom{m}{D}\binom{p}{d}^{m-D}\binom{p}{d-1}^{D} \Big/ \binom{p}{d}^{m} \le p\left(\frac{emd}{D(p-d+1)}\right)^{D} \le p/2^{D}$$

for $D \ge 4emd/p$, and $d \le p/2$. □

The following technical lemma is the heart of the argument that establishes the existence of generalized expanders.

Lemma 6 *For all m, p, n, d, g, r, and λ satisfying (i) $m \ge p \ge n$, (ii) $p \ge 2d$, (iii) $g \ge 2$, (iv) λ a constant less than 1, (v) $r \ge 2$, and*

$$(vi)\quad \mu \triangleq e\lambda d^3(m/p)\left(\frac{(n/p)^2}{g^r}\right)^{(1/4)(1-\lambda)(d/r)} < 1/2$$

then at most $2\mu|B_{m,p,d}|$ of the graphs in $B_{m,p,d}$ are not $(m,p,\lambda,d,d/r,\sigma)$-generalized expanders, where $\sigma(k) = (n/p)/dg^{d/k}$.

Proof For each G that is not a $(m,p,\lambda,d,d/r,\sigma)$-generalized expander, there is a witness (S,W,F), where $S \subseteq U$ of size at most $\sigma(k)p$, $W \subseteq V$ of size at most $\lambda k|S|-1$, and $F \subseteq E(S)$ is a k-bundle such that $\Gamma_F(S) \subseteq W$. (Here $E(S)$ represents the set of edges in E incident on the vertices in S.)

Let $s \triangleq |S|$. Straightforward combinatorial arguments show that that there are

$$\binom{m}{s},\ \binom{p}{\lambda ks-1},\ and \binom{\lambda ks}{k}^{s}$$

ways of choosing S,W, and F respectively. Also a fraction

$$\binom{p}{d}^{m-s}\binom{p}{d-k}^{s}\Big/\binom{p}{d}^{m}$$

of graphs in $B_{m,p,d}$ have (S,W,F) as a witness.

Thus f, the fraction of graphs in $B_{m,p,d}$ that are not $(m,p,\lambda,d,d/r,\sigma)$-generalized expanders, may be bounded by the following expression

$$f \le \sum_{k=c}^{d/2}\sum_{s=1}^{\sigma(k)p}\binom{m}{s}\binom{p}{\lambda ks}\binom{\lambda ks}{k}^{s}\binom{p}{d}^{m-s}\binom{p}{d-k}^{s}\Big/\binom{p}{d}^{m}.$$

In order to bound the quantity f we will make use of the following inequalities

$$\binom{m}{s} \le (em/s)^s,$$

$$\binom{p}{\lambda ks} \le [ep/(\lambda ks)]^{\lambda ks}, \textit{ and}$$

$$\binom{\lambda ks}{k}\binom{p}{d-k}\Big/\binom{p}{d} \le \binom{d}{k}\left\{\frac{2\lambda ks}{p}\right\}^{k}.$$

The bound on f simplifies to

$$\begin{aligned}
f &\le \sum_{k=c}^{d/2}\sum_{s=1}^{\sigma(k)p}\left(\frac{em}{s}\right)^{s}\left(\frac{ep}{\lambda ks}\right)^{\lambda ks}\left[(ed/k)^{k}\left\{\frac{2\lambda ks}{p}\right\}^{k}\right]^{s} \\
&\le \sum_{k=c}^{d/2}\sum_{s=1}^{\sigma(k)p}\left[\left(\frac{em}{s}\right)\left(\frac{\lambda ks}{p}\right)\left\{A^{k}(\lambda)(d/k)^{k}\left(\frac{ks}{p}\right)^{k-1-\lambda k}\right\}\right]^{s} \\
&\qquad (\text{where } A(\lambda) \triangleq 2\lambda^{1-\lambda}e^{1+\lambda}) \\
&\le \sum_{k=c}^{d/2}\sum_{s=1}^{\sigma(k)p}\left[\left(\frac{e\lambda mk}{p}\right)\left\{A(\lambda)(d/k)\left(\frac{ks}{p}\right)^{(1/2)(1-\lambda)}\right\}^{k}\right]^{s}
\end{aligned}$$

under the assumption that $k \ge 2/(1-\lambda)$.

The expression in square brackets is increasing in s and so by substituting $\sigma(k)p$ for s and interchancing the order of the sums f may be bounded as follows.

$$
\begin{aligned}
f &\leq \sum_{k=c}^{d/2} \sum_{s=1}^{\sigma(k)p} \left[\left(\frac{e\lambda mk}{p}\right) \left\{ A(\lambda)(d/k) \left(\frac{k(n/p)}{dg^{d/k}}\right)^{(1/2)(1-\lambda)} \right\}^{k} \right]^{s} \\
&\leq \sum_{s=1}^{p} \left[\sum_{k=c}^{d/2} \left(\frac{e\lambda mk}{p}\right) \left\{ A(\lambda)(d/k) \left(\frac{k(n/p)}{dg^{d/k}}\right)^{(1/2)(1-\lambda)} \right\}^{k} \right]^{s}.
\end{aligned}
$$

Now observe that $A(\lambda) = 2\lambda^{1-\lambda}e^{1+\lambda} \leq 2e^2 < 16$. Straightforward arithmetic manipulations show that if $g \geq 2^{12/(1-\lambda)}$ then

$$g^{(1/4)(1-\lambda)(d/k)} \geq 2^{3(d/k)} \geq 4^{d/k}2^{d/k} \geq 16(d/k).$$

Thus the term in square brackets simplifies to

$$\sum_{k=d/r}^{d/2} \left(\frac{e\lambda mk}{p}\right) \left\{\frac{n/p}{g^{d/2k}}\right\}^{(1/2)(1-\lambda)k} \leq e\lambda d^3 (m/p) \left((n/p)^2 / g^r \right)^{(1/4)(1-\lambda)(d/r)}$$

This last expression is exactly μ which is at most 1/2 by assumption, and so

$$f \leq \sum_{s=1}^{\infty} \mu^s \leq 2\mu < 1.$$

This concludes the proof. □

The combination of the preceding two lemmas yields the following theorem

Theorem 5 *For all m, p, n, g, r, and λ satisfying (i) $m \geq p \geq n$, (ii) $g \geq 2$, (iii) λ a constant less than 1, and (iv) $r \geq 2$, there exists a constant β depending only on λ such that for all*

$$d \geq \beta \frac{r \log(m/p)}{[r \log g + \log(p/n)]}$$

there exists balanced $(m, p, \lambda, d, d/r, \sigma)$-generalized expander in $B_{m,p,d}$ with $\sigma(k) = (n/p)/dg^{d/k}$.

As a corollary we have the following.

Corollary 1 *For every n, m, g and λ, with $m \geq n$, $g \geq 2$ and λ a constant less than 1, there is a constant β depending only on λ such that for every $c \geq \beta \log(m/n)/\log g$ there exists a $(m, n, \lambda, 2c-1, c, 1/gc)$-generalized expander.*

Upfal and Wigderson[UW87] established the existence of a family of $(m, n, \lambda, 2c-1, c, 1/(2c\text{-}1))$-generalized expanders for every $\lambda \leq 1/2$, with $c \geq \beta \log m$. These generalized expanders and similar ones define the memory organizations used in [UW87,AHMP87,LPP87]. They can essentially be derived from the corollary by choosing $g = 2$. The memory organization of [HP89] is essentially a $(m, n, \lambda, 2c-1, c, 1/gc)$-generalized expander with $g = n^{\epsilon}$ for some $\epsilon > 0$.

Theorem 5 is the only nonconstructive result employed in this thesis. In would clearly be more desirable to have a constructive description of these graphs in terms of some easily computable functions. By the remarks made earlier about the relationship between generalized expanders and concentrators, an explicit construction for generalized expanders would imply a constructive description of concentrators, an open graph-theoretic problem. The combinatorial arguments employed in Lemmas 5 and 6 imply that the overwhelming majority of graphs in $B_{m,p,d}$ are balanced $(m, p, \lambda, d, c, \sigma)$-generalized expanders for sufficiently large values of d. This implies that a graph picked from $B_{m,p,d}$ is overwhelmingly likely to be a balanced $(m, p, \lambda, d, c, \sigma)$-generalized expander.

The following lemma gives a proof of the assertion made earlier that the generalized expander property guarantees that corresponding to each set S of variables is a $(\lfloor d/2 \rfloor + 1, q)$-sheaf with tolerable q. The next section provides the tools need to construct such a sheaf.

Lemma 7 *If $G = (U, V; E)$ is a $(m, p, \lambda, d, c, \sigma)$ generalized expander, then for any $Z \subseteq U$ of size at most p there is a $(\lfloor d/2 \rfloor + 1, q)$-sheaf, with $q \triangleq (1/\lambda)\lceil |Z|/\sigma(\lfloor d/2 \rfloor + 1)p \rceil$.*

Proof Let $\lfloor d/2 \rfloor + 1$ be abbreviated to k. Let Z be a subset of U of size at most n. Partition Z into $K \triangleq \lceil |Z|/\sigma(k)p \rceil$ subsets $Z_1, Z_2, \ldots, Z_K$ each of size at most $\sigma(k)p$.

Consider some Z_i, and let F_i be the set of edges incident on the vertices Z_i. Let $\widehat{G}$ be a bipartite graph obtained from G, by replacing each $u \in U$ with $\widehat{u^1}, \ldots, \widehat{u^k}$, each $v \in V$ with $\widehat{v^1}, \ldots, \widehat{v^{k/\lambda}}$, and each edge $(u, v) \in E$ with the edges of the set $\{(\widehat{u^i}, \widehat{v^j}) : 1 \leq i \leq k \text{ and } 1 \leq j \leq k/\lambda\}$. Corresponding to the subset Z_i in U is a subset $\widehat{Z_i}$ in $\widehat{U}$. Let S' be any subset of $\widehat{Z_i}$, and let S be the subset of Z_i defined by

$$S \triangleq \{u \in Z_i : \widehat{u^j} \in \widehat{S'} \; for\; some\; 1 \leq j \leq k\}.$$

Notice that $|S'| \leq k|S|$. The set S' has at least

$$(k/\lambda)(\lambda|S|) \geq |S'|$$

neighbours in $\widehat{V}$ since $|S| \leq \sigma(k)p$. By Hall's Theorem (see [Har69] for example) there exists a matching for $\widehat{Z_i}$ in $\widehat{G}$. The set of edges F_i in G corresponding to this matching constitutes a $(k, 1/\lambda)$-sheaf for Z_i. The union of the sheaves for $Z_1, \ldots, Z_K$ is a $(k, (1/\lambda)K)$-sheaf for Z in G. □

3.3 Properties of Generalized Expanders

In this section we explore some of the properties of generalized expanders that allow processors to select copies to minimize congestion during the memory access algorithm.

Let $G = (U, V; E)$ be an $(m, p, \lambda, d, c, \sigma)$-generalized expander and let H be a subgraph of G induced by some set of variables Z. We will show that by discarding a carefully selected set of edges H may be transformed in a $(d-c, q)$-subgraph H' of H for some suitably modest value of q. The edges are discarded in stages called culling steps. Informally, each culling step deletes a set of edges chosen so that the number of edges remaining incident on "congested" vertices (those with degree greater than q) is sharply reduced. After a suitable number of such steps no remaining edges are incident on congested vertices.

Definition 4 *A vertex $v \in V$ is said to be* q-congested *in H if $deg(v, H) > q$. An edge (u, v) is q-congested if v is. A vertex $u \in U$ is* (k, q)-active *if more than k of its neighbours are q-congested.*

Intuitively, the active vertices correspond to those variables with many copies trapped in congested modules.

Definition 5 *A* (k, q)-culling *applied to H deletes from H all q-congested edges incident on vertices that are not (k, q)-active.*

Let $Cull_{k,q}(H)$ denote the graph resulting from the application of a (k, q)-culling step to H.

For a given graph $H = (U_H, V_H; E_H)$ the vertices in U_H may be partitioned into three sets depending on the number of incident q-congested edges. Let $C_q(H)$ denote the set of q-congested edges in the subgraph H, and let $cong(u, H)$ denote the number of edges in $C_q(H)$ incident on the vertex u. Let

$$\begin{aligned} A_{k,q}(H) &\triangleq \{u \in U : k < cong(u, H) \le d\}, \\ N_{k,q}(H) &\triangleq \{u \in U : 1 \le cong(u, H) \le k\}, \text{and} \\ O_{k,q}(H) &\triangleq \{u \in U : cong(u, H) = 0\}. \end{aligned}$$

Let $\hat{H}$ represent $Cull_{k,q}(H)$. Notice that a culling step deletes only edges belonging to vertices in $N_{k,q}(H)$, and each vertex in this set loses all its q-congested edges (of which it has at most k by definition). Thus $N_{k,q}(H) \cup O_{k,q}(H) = O_{k,q}(\hat{H})$, and $A_{k,q}(\hat{H}) \subseteq A_{k,q}(H)$. One consequence of the expander property of the graph G is that $|A_{k,q}(\hat{H})|$ is significantly smaller than $|A_{k,q}(H)|$ is.

The following lemma is essentially a rephrasing of the expansion property of the generalized expander.

Lemma 8 *For any $Z \subseteq U$ and for every k-bundle F of Z $|\Gamma_F(Z)| \leq \lambda k\sigma(k)p$ implies that $|Z| \leq |\Gamma_F(Z)|/\lambda k$.*

This may be used to place a bound on the number of vertices that survive a (k, q)-culling step.

Lemma 9 *If $k \geq c$, and if $|A_{k,q}(H)| \leq (\lambda kq/d)\sigma(k)p$, then*

$$|A_{k,q}(\mathrm{Cull}_{k,q}(H))| \leq (d/\lambda kq)|A_{k,q}(H)|.$$

Proof Let H be a subgraph of G such that $|A_{k,q}(H)| \leq (\lambda kq/d)\sigma(k)n$, and let $\hat{H}$ denote the subgraph $Cull_{k,q}(H)$.

Let C denote the set of q-congested vertices in $\hat{H}$. The culling step removed all edges in H incident on the vertices of C except for those belonging to $A_{k,q}(H)$. Thus the size of C may be bounded as follows

$$|C| \leq d|A_{k,q}(H)|/q \leq \lambda k\sigma(k)p.$$

where we have used the assumed bound for $|A_{k,q}(H)|$.

Since each (k, q)-active vertex has at least $k + 1$ neighbours that are q-congested, C must contain some k-shadow of $A_{k,q}(\hat{H})$. Therefore

$$|A_{k,q}(\hat{H})| \leq |C|/\lambda k \leq (d/\lambda kq)|A_{k,q}(H)|,$$

which establishes the lemma. □

Consider a sequence of (k, q)-cullings applied to a subgraph $H = G(Z)$. Let $Cull^{(i)}_{k,q}(H)$ denote the state of H after the i^{th} culling. With each culling the number of (k, q)-active vertices shrinks according to Lemma 9, while the number of inactive vertices grows. In fact the size of the active component of $Cull^{(i)}_{k,q}(H)$ is at most $(d/\lambda kq)^i|Z|$, by Lemma 9. Also at most k edges belonging to any variable are deleted during the sequence of cullings, since a variable loses at most k edges during the first culling step after it has become inactive and loses no further edges in subsequent cullings. Moreover, the inactive component of $Cull_{k,q}^{(i)}(H)$ has congestion at most q. These observations may be summarized as follows.

Lemma 10 *With the above notation, if $Z \subseteq U$ has size at most $(\lambda kq/d)\sigma(k)p$, then $\mathrm{Cull}_{k,q}^{(i)}(G(Z))$ contains a $(d-k, q)$-subgraph for all but at most $(d/\lambda kq)^i |Z|$ of the variables in Z.*

Corollary 2 *Let $G = (U, V; E)$ be a $(m, p, \lambda, d, c, \sigma)$-generalized expander, let $Z \subseteq U$ be of size at most p, then for all $q \geq (d/\lambda k)\lceil |Z|/\sigma(k)p \rceil$ and for all k such that $c \leq k \leq \lceil d/2 \rceil - 1$, the number of (k,q)-culling steps required to transform $G(Z)$ into a $(d-c, q)$-subgraph is at most* $\log n / \log(\lambda(k/d)q)$.

We conclude this chapter with a number of observations that will prove useful in later chapters.

The first observation is that during a sequence of (k, q)-culling steps applied to a subgraph H, the i^{th} culling and all subsequent cullings affect only the edges belonging to vertices in $A_{k,q}(Cull_{k,q}^{(i)}(H)) \cup N_{k,q}(Cull_{k,q}^{(i)}(H))$. All remaining edges incident on vertices in $O_{k,q}(Cull_{k,q}^{(i)}(H))$ can not be q-congested by definition and so will never be deleted. Thus the size of the fragment of the graph that a culling algorithm is actively concerned with decreases sharply with each culling step.

The second observation is that the fate of a vertex u in a graph H during a (k, q)-culling step (*i.e.* whether or not it remains active afterwards) depends only on "nearby" edges. To formalize this idea, for some $v \in V$ let $\Lambda(v)$ denote the set of neighbours of v in G, and let $v_u^1, \ldots, v_u^d$ denote the neighbours of

$u \in U$. For each edge (u, v_u^i) let

$$a_H^{(t)}(u, v_u^i) \triangleq \begin{cases} 1 & if\, (u, v_u^i)\, is\, in Cull_{k,q}^{(t)}(H) \\ 0 & otherwise \end{cases}.$$

Now it is easy to see that $a_H^{(t+1)}(u, v_u^i)$ depends on the values of $a_H^{(t)}(w, v_w^i)$ for all vertices w such that $v_u^k = v_w^j$ for some k and some j *i.e.* $w \in \Lambda(v_u^k)$. Whether or not u is (k, q)-active in $Cull_{k,q}^{(t+1)}$ is a function of $a_H^{(t)}(w, v_w^j)$ for all vertices $w \in \Lambda(v_u^k)$, and for all j and k in the range 1 to d.

Chapter 4

An Optimal Simulation For Large Memory

In this chapter, we present our first simulation algorithm (which also appeared in [HB88]) to simulate an (m,n)-PRAM on a BDN with n nodes. For the special case $m \geq n^{1+\epsilon}$ (ϵ a positive constant), this algorithm achieves the time bound specified in Theorem 1, that is $T = O(\log m \log n / \log\log n)$. As it will be shown in Chapter 8 this performance is optimal for any online point-to-point protocol, if $\Omega(n^{1+\epsilon}) \leq m \leq 2^{(\log n)^{O(1)}}$. A more sophisticated simulation will be developed in Chapter 5, which improves on those of this chapter for $m < n^{1+\epsilon}$.

4.1 Introduction

The main result of this chapter is as follows.

Theorem 6 *There is an n-node BDN (V, E_{sim}) and a deterministic algorithm that can simulate an arbitrary step of an (n,m)-PRAM ($m \geq n$) in*

$$O(\log m \log n / \log\log n)$$

time.

When m is in the range $n^{1+\epsilon} \leq m \leq 2^{(\log n)^{\alpha}}$, the above upper bound is within a constant factor of lower bound of Theorem 2.

The memory access algorithm has the following overall structure. The variables are accessed in stages called *decimations*. Each decimation accesses up to some number q copies from each node. With the appropriate choice of memory organization this succeeds in accessing a quorum of copies for all but a small fraction of the variables. The small fraction, called the *survival factor*, is proportional to $1/q$. At any point in the execution of the algorithm, the variables can be partitioned into two sets: those for which a quorum copies have been accessed (called the *dead* variables), and the remaining ones (*live* variables). Dead variables play no further role in the simulation since enough copies have been accessed to effect either a read or write. For the live variables, more copies need to be accessed.

By way of contrast, the protocols of [UW87] and [AHMP87] are built around a *halving* procedure which, although implemented differently, has the same effect as a decimation with survival factor of at most $1/2$.

Intuitively, the advantage of decimation over halving can be understood as follows. From standard considerations on the diameter of an n-node BDN it is clear that any implementation of halving will require $\Omega(\log n)$ time in the worst case. However it will be shown in the next section that $O(\log n)$ time is essentially sufficient to decimate a (sufficiently small) set of variables. A consequence of the memory organization chosen here is that since the survival factor for decimation is significantly less than that of a halving, decimations can kill more variables than halving in the same amount of time. A similar observation is also employed in the scheme of [LPP87].

4.2 The Memory Access Algorithm

The memory organization is a balanced $(m, n, \lambda, d, d/2, \sigma)$-generalized expander $G = (U, V, E)$, with $\sigma = 1/gd$, and λ a constant less than one. By the Corollary 1 of Chapter 3, such a memory organization exists with $d = \beta \log(m/n)/\log g$. The value of g will be chosen later to balance the work needed to extract the relevant information from the memory modules with the work needed to route this information to the intended processors.

Since G is balanced, at most $O((m/n)d)$ items are mapped to any individual node. Within each node the items are stored in a static dictionary, as described in [FKS84]. The essential property of such a dictionary is that N items may be stored in $O(N)$ storage, and any one may be retrieved in $O(1)$ time given its name. In this chapter we will assume that each node keeps a copy of a table giving the structure of G in its local memory. The techniques of Chapter 7 will remove this assumption. The structure of the BDN will be specified later.

We assume that the instruction to be executed by PRAM processor P_i^{PRAM} at a given step is available to the corresponding BDN processor P_i at the beginning of the simulation of this step.

For simplicity we assume that all the accesses in a given step are of the same type (read or write), and that no two accesses refer to the same memory cell. Both assumptions can be removed by established techniques [Vis83] without altering the stated time bounds, as already noted in [UW87].

Let Z be the set of PRAM variables to be accessed in a given PRAM step. The goal of the simulation is to access a majority of the copies of each variable in Z. The simulation is described here for a read step, a write step being handled in a very similar manner.

ParRead

I For each processor P_i, generate u_i, the name of the variable it wishes to read. For each u_i and each $j \in \{1, \ldots, d\}$

generate a *request packet* labelled with an origin P_i, a destination $\gamma_j(u_i)$, the label u_i itself, and a field to store the value and the timestamp of the copy of u_i contained in node $\gamma_j(u_i)$. (Recall that the $\gamma_j(u_i)$ represent the nodes holding copies of the variable u_i. The destination of a request packet is the node holding the corresponding copy.)

II Deliver at least $\lfloor d/2 \rfloor + 1$ request packets from each origin to their destinations, and return these packets to their origins with the values and the timestamps.

III At P_i, select the value from the returned packet(s) with the most recent timestamp as the result of read(u_i).

Steps I, and III are straightforward and each accomplished in $O(d)$ time. We focus our attention on Step II. The latter will be implemented as a sequence of decimations, the first one applied to the set Z, and the remaining decimations each applied to the set of variables left alive by the previous decimation.

4.3 Implementation of *Decimate*

In this subsection we describe the implementation of a procedure *Decimate(S, q)* with the following properties: (i) S is a variable parameter and represents a subset of U of size at most n distributed among the nodes of the BDN, with at most one element per node; (ii) the second argument q is a value parameter of type integer; (iii) the execution of *Decimate(S,q)* will cause all the packets destined for any non q-congested node v (a node with at most q packets destined for it) to reach that node and to be returned to their respective origins, with the appropriate values and timestamps (q-congested nodes do not respond to any requests), (iv) the value of parameter S is modified by the execution of *Decimate(S,q)* so that, upon termination, S represents the set of live variables.

The BDN underlying the algorithm has the structure (V, E_{route}), the BDN presented in Chapter 2. Recall that (V, E_{route}) included the n-node sorting network (V, E_{AKSL}) of [AKS83,Lei85], and a complete binary tree on n nodes (V, E_{tree}) as subgraphs.

The overall idea is as follows. It is assumed that the request packets corresponding to the copies of a set of variables S are distributed evenly among the nodes of the BDN. A sorting step is used to group the packets with a common destination together. Once grouped in this way a parallel prefix operation may be used to determine which destinations have at most q packets destined for them, and to select all packets destined for these "uncongested" nodes. Routing the selected packets to their respective destinations is an instance of the $(\lceil |S| d/n \rceil, d, q)$-routing problem. Once the packets have been loaded with the appropriate values they may be routed back to their origins by retracing their steps. The details follow.

Decimate(S, q)

(1) Sort the request packets by destination.

(2) For each destination v, determine d_v the number of packets that have v as their destination. If $d_v \leq q$ then mark all requests destined for v *selected.* Otherwise select none.

(3) Route the selected packets to their respective destinations.

(4) Load values and timestamps into request packets.

(5) Route the requests packets back to their origins tracing the paths taken in Steps 1 and 3 backwards.

(6) Let each origin count the number of packets received during Step 5 and if they are at least $\lfloor d/2 \rfloor + 1$ in number, retain the value of the packet(s) with the most recent timestamp and remove the variable from S.

Lemma 11 Decimate(S,q) *can be implemented on the BDN* (V, E_{route}) *with running time* $O(d + q + \log n \lceil d|S|/n \rceil)$.

Proof Firstly Steps 4, and 6 are sequential computations local to a given node and take $O(q)$, and $O(d)$ time respectively.

Step 2 can be reduced to prefix computations [KRS85] on sequences of length $d|S|$ which can be executed on (V, E_{Tree}) in $O(\log n + \lceil d|S|/n \rceil)$ time.

Step 1 consists of sorting a distributed set of size no larger than $d|S|$ and with at most d elements per node. This takes $O(d + \log n \lceil d|S|/n \rceil)$ time.

Step 3 involves routing a distributed set of size no larger than $d|S|$ and with at most d packets per source, and at most q packets per destination. By Theorem 4, this takes $O(d + q + \log n \lceil d|S|/n \rceil)$.

Step 5 involves retracing the steps taken during Steps 1 and 3. If during these steps, each node recorded the arrival and departure (the time and the edge involved) of each packet which passed through it, this information will allow packets to retrace their steps during Step 5. The details are straightforward.

By summing the contributions of all the steps it is easy to see that the stated running time is achieved. □

In the terminology of Chapter 3, the variables left alive after *Decimate*(S, q) are those with a quorum of copies confined in q-congested nodes *i.e.* these are exactly the set of $(\lfloor d/2 \rfloor + 1, q)$-active vertices in the subgraph $G(S)$ of G induced by the set S. The following lemma, which bounds the number of variables left alive after *Decimate*(S, q), is a consequence of Lemma 9 of Chapter 3 and the fact that there can be at most $(d/q)|S|$ q-congested nodes $G(S)$.

Lemma 12 *With the above notation, if the memory organization is a* $(m, n, \lambda, d, d/2, 1/gd)$*-generalized expander* $G = (U, V; E)$ *and if*

$$q \geq (2/\lambda)(|S|/n)dg^2,$$

then the set L of variables remaining alive after Decimate(S,q) *satisfies the relation*

$$|L| \leq (2/\lambda q)\,|S|. \tag{4.1}$$

4.4 Accessing the Variables

Step II of the memory access protocol can be written as follows.

$S \leftarrow X$;
for $h \leftarrow 1$ **to** H **do**
    Decimate(S, q_h)

The values of $q_1, q_2, \ldots, q_H$ and H will be specified later so that after the H^{th} iteration S is empty, *i.e.* all the variables in X have been accessed.

We now turn our attention to the running time of Step II of the memory access protocol consisting of a sequence of H decimations with parameters $q_1, q_2, \ldots, q_H$. Let L_i be the set of live variables after the i^{th} decimation.

For the first decimation, applied to the set S of PRAM variables, choose $q_1 = (2/\lambda)dg^2$. By Lemma 12 $|L_1| \leq (2/\lambda q_1)|S|$, and by Lemma 11, *Decimate(S,q_1)* runs in $O(dg^2 + d\log n)$ time.

For the remaining decimations we choose $q_2 = q_3 = \ldots = q_H = g$. By Lemma 12 for $i \geq 2$, $|L_i| \leq (2/\lambda g)|L_{i-1}|$. Thus, for $i \geq 2$, $|L_i| < (2/\lambda g)^{i-1}|L_1| \leq (2/\lambda g)^i n$. Choosing $H = \lceil \log n/\log(\lambda g/2)\rceil$ ensures that $L_H = \emptyset$.

The total time for Step II, and indeed for the entire simulation is

$$O(dg^2 + d\log n + (d + g + \log n)(\log n/\log g)).$$

Choosing $g = \sqrt{\log n}$ and recalling that $d = \beta\log(m/n)/\log g$, we arrive at the timebound stipulated in Theorem 6.

Notice that in our protocol obeys the point-to-point communication assumption since each message dispatched to read or update a copy is treated as a separated entity by the routing network: requests are generated only at their origins, while the other processors only pass the requests from their origins to their respective destinations and back.

Chapter 5

Simulations For Small Memory

This chapter develops an algorithm that can simulate any step of an (n, m)-PRAM on a particular n-node bounded degree network within the time bound stipulated in Theorem 1. The algorithm will be modified in Chapter 7 to meet the space bounds specified in that theorem.

The algorithm is presented in Sections 5.1 to 5.5. Section 5.6 extends the result to the case where p, the number of nodes on the simulating BDN, is greater than n, the number of PRAM processors. It will be shown that simulations with faster running times than Theorem 1 are possible in some cases. An earlier version of these results appears in [Her89a].

5.1 The Memory Access Protocol

The results of this chapter are presented in terms of the BDN model. The simulation of this chapter is fairly independent of the underlying machine in the sense that it relies on a handful of communication primitives such a sorting and the generalized routing algorithm discussed in Chapter 2. The simulation can be ported to any machine that supports these primitives though the simulation will not necessarily be as efficient as the one presented here. In the next chapter it will be shown that all of the requisite communication primitives are supported very efficiently on the augmented mesh-of-trees architecture introduced in Chapter 1.

Recall that the BDN (V, E_{route}) underlying the generalized routing algorithm of Theorem 4 includes a n-node AKS sorting network (V, E_{AKSL}), and a complete binary tree (V, E_{tree}) with n nodes.

If $m = O(n \log n)$ then the following simple simple simulation is sufficient. Distribute the variables evenly among the n nodes of the BDN. By Theorem 4 any n-tuple of variables may be accessed in $O(\log n)$ time. It will be assumed that $m \geq n \log n$ for the remainder of this chapter.

The memory organization G is chosen to be a n $(m, n, \lambda, d, d/r, \sigma)$-generalized expander, with λ a constant less than one, $\sigma(k) \triangleq 1/dg^{d/k}$, and $g \triangleq \sqrt{\log n}$. The value of the parameter r will be specified later. By Theorem 5 such a G exists with

$$d = \beta \log(m/n)/\log\log n$$

copies per variable, for some constant β. It is assumed for now that each node of the BDN stores, in its local memory, a table giving the locations of the copies of each variable. In the Chapter 7 this restriction will be removed.

Since both reading and writing involve accessing $\lfloor d/2 \rfloor + 1$ copies of a variable, the memory access algorithms for reading and writing are virtually identical. Assume that $Z \triangleq \{u_1, \ldots, u_n\}$ is the set of variables to be read. It is assumed that u_i is known to P_i^{BDN} at the start of the simulation step. The overall structure of the reading protocol is as follows. Let

$$q_G \triangleq (2/\lambda)dg^2. \tag{5.1}$$

ParRead

I For each $1 \leq i \leq n$, let P_i^{BDN} generate d packets, each packet is labelled with its *origin* P_i^{BDN}, and the name of one of u_i's edges. Let H represent the distributed set consisting of all these packets, and notice that H represents the edges of the subgraph $G(Z)$.

II Construct a $(\lfloor d/2 \rfloor + 1, q_G)$-subgraph H_Z from H.

III Route the packets representing the edges in H_Z to their destinations. (The destination of a packet labelled with the edge (x, y) is the node y.) Load each packet with the appropriate value and timestamp and return all packets to their respective origins.

IV Let each processor inspect the $\lfloor d/2 \rfloor + 1$ packets received during the previous step and select the value with the most recent timestamp.

Steps I and IV can both be executed locally at each BDN node in $O(d)$ time. Step III can be accomplished in $O(q_G + d \log n)$ time using the (nd, d, q_G)-routing algorithm of Chapter 2. All that remains is to specify the implementation and running time of Step II.

The construction of H_Z is in two parts. The first part constructs a $(\lfloor d/2 \rfloor + 1, q_G)$-subgraph for $Z' \subseteq Z$ containing all but a handful of the variables in Z, by executing a small number of $(\lceil d/2 \rceil - 1, q_G)$-culling steps. The construction of the $(\lfloor d/2 \rfloor + 1, q_G)$-subgraph for the remaining variables $Z'' = Z - Z'$ is achieved by applying a sequence of *refinements* to $G(Z'')$ the subgraph of G induced by the variables in Z''. Each refinement reduces the congestion of the subgraph by discarding a carefully selected (d/r)-bundle of edges. The reason for the two part construction may be understood as follows. Part one of the algorithm performs a number of $(\lceil d/2 \rceil - 1, q_G)$-culling steps, and relies on the expansion of $(\lceil d/2 \rceil - 1)$-bundles, which holds for sets of size at most $n/(dg^2)$. The second part of the algorithm relies on the expansion of (d/r)-bundles, which holds only for much smaller sets. The role of the first part is to reduce the size of the problem to the point where the more powerful techniques may be applied.

Section 5.2 presents two procedures *SimpleCull* and *TreeCull* that form the basis for the construction of H_Z. Section 5.3 develops the procedure *Refine* based on *SimpleCull* and *TreeCull* that effects the refinements mentioned above. Section 5.5 describes the overall construction of the subgraph H_Z in terms of these procedures.

5.2 Two Culling Procedures

Two procedures *SimpleCull*, and *TreeCull*, which are employed in the construction of the subgraph H_Z, are presented in this section. Each of these procedures performs a sequence of culling steps.

With respect to a particular graph $H = (U_H, V_H; E_H)$, and integer q, we say that a vertex $u \in U_H$ is *q-awake* if one or more of its edges in H are q-congested. When the value q is clear from the context we will abbreviate the term *q-awake* to *awake*; vertices in U_H that are not awake are said to be *asleep*.

Notice that the set of q-awake vertices includes all the (k, q)-active vertices. A (k, q)-culling deletes all q-congested edges incident on all q-awake vertices except those that are (k, q)-active.

The procedure *SimpleCull*(H, A, k, q, t) takes two variable parameters H representing a subgraph and A representing the set of q-awake vertices in H, and three value parameters k, q, and t. The procedure performs t (k, q)-culling operations on the subgraph H. It is assumed that the subgraph H is represented by a balanced distributed set of packets labelled with the edges of H. At the conclusion of the execution of the subroutine the parameter H contains $Cull_{k,q}^{(t)}(H_{init})$, where H_{init} is the initial value of the parameter H. The parameter A contains all the q-awake vertices in $Cull_{k,q}^{(t)}(H_{init})$.

The implementation of *SimpleCull* is essentially identical to the procedure

Decimate of Chapter 4, except that no information is physically extracted from the nodes of the BDN. An analysis that parallels that of Chapter 4 yields the following result. (Let $|H|$ denote the number of edges in the subgraph H.)

Lemma 13 *There is a deterministic algorithm implementing* SimpleCull *that runs in* $O(\{\lceil |H|/n \rceil + t\} \log n)$ *time on an* n*-node BDN with the structure* $(V, E_{route} \cup E_{tree})$.

SimpleCull requires a global communication step to accomplish each culling. However, as observed at the conclusion of Chapter 3, the fate of a vertex u after a culling step depends only on the state (deleted or present) of the " nearby " edges, *i.e.* edges incident on the neighbours of u. Thus computing the effect of a culling on a particular vertex $u \in U$ requires only *local* information about the state of edges around u.

This observation will be exploited in *TreeCull* to compute the effect of a number of cullings using only local communication for each. Informally, the idea is to devote a suitably labelled subtree of the network to each awake vertex u to represent the state of a small neighbourhood of u in the subgraph H of G. The labelling is *locally consistent* with the structure of G in the following sense: the children of a node with label x are labelled with the neighbours of x in G. Global communication is needed only to set up the trees. The number of cullings possible with this technique is related to the depth of the tree. Having completed a number of culling steps in this way, all but a few of the vertices originally awake will have fallen asleep. The process may be repeated and, since there are fewer vertices awake, larger trees can be devoted to each thus enabling a greater number of culling steps to be executed per unit time. (The idea of using trees to represent small neighbourhoods of a network to mask network latency appears in [Mey86].)

The description of the algorithm involves two graphs: G_{MO} representing the memory organization, and G_{BDN} representing the structure of the underlying

BDN. To avoid confusion the term *vertex* will be used only in connection with the graph G_{MO} and the term *node* will be used in connection with G_{BDN}.

Conceptually, what is needed is a tree with nodes with either d children (to represent vertices in U) or $\Delta(H)$ children (to represent the vertices of V). In the implementation such a tree will be mapped into a subtree of the binary tree (V, E_{tree}) of the underlying BDN.

A (f_1, f_2, h)-supertree is a binary tree of height $h\delta$ where $\delta \triangleq \lceil \log f_1 \rceil + \lceil \log f_2 \rceil$. The nodes at height $l\delta$, for $0 \leq l \leq h$, are called U-nodes at *pseudoheight* l. Nodes at heights $l\delta - \lceil \log f_1 \rceil$, for $1 \leq l \leq h$ are referred to as V-nodes. For a U-node x at pseudoheight h', the V-nodes descendent from x at height $h'\delta - \lceil \log f_1 \rceil$ are called its V-offspring. The U-offspring of a V-node are similarly defined.

A $(d, \Delta(H), h)$-supertree will be employed for each u that is awake, the vertices of which will be labelled with the names of vertices of U and V as follows. (Given a labelled tree let l_x denote the label of node x.) A supertree is said to be (u, H)-*labelled* if (i) the root is labelled u. (ii) the V-offspring of each U-node x are labelled with the neighbours of l_x in H if l_x is awake, and are unlabelled otherwise, (iii) the U-offspring of each labelled V-node y are labelled with the neighbours of l_y in H, and (iv) the children of unlabelled nodes are unlabelled.

Notice that portions of an (u, H)-tree may be unlabelled. If d or $\Delta(H)$ are not powers of two, and if a vertex has fewer edges than allowed then some of subtrees of the associated node will be unlabelled. Also, subtrees rooted at U-nodes labelled with a vertex that is asleep will not be labelled. Nodes with no labelled descendants together with the leaves are referred to as the *frontier* nodes of the labelled tree.

The procedure *TreeCull*$(\mathcal{F}, H, A, k, q)$ takes two variable parameters H representing a subgraph and A, and three value parameter, two integers k and q

and $\mathcal{F}$ a forest of $(d, \Delta(H).h)$-supertrees. Each tree Θ_u in $\mathcal{F}$ is (u, H)-labelled for some $u \in A$. The procedure transforms the initial value H_{init} of the parameter H into $H_{final} = Cull^{(h)}_{k,q}(H_{init})$ by performing a sequence h of (k, q)-cullings. At the conclusion the parameter A contains all the q-awake vertices in H_{final}.

The procedure *TreeCull* may be implemented as follows.

TreeCull$(\mathcal{F}, H, A, k, q)$

(i) Mark each frontier node x with the state (awake, asleep) of the associated vertex l_x in H.

(ii) Mark the non-frontier nodes in a bottom up fashion according to the following rules: for each V-node mark it congested if more than q of its U-offspring are marked awake, and mark it uncongested otherwise: for each U-node mark it awake if more than k of its V-offspring are marked congested. and mark it asleep otherwise.

(iii) For each $\Theta_u \in \mathcal{F}$ whose root is marked asleep do the following. Check whether u is (k, q)-active and if not discard all the q-congested edges of u from H. These edges can be determined by examining the markings of the children of the root.

(iv) Let A be the set of q-awake vertices with respect to the updated H.

The running time may be analysed as follows. Step (i) can be implemented by interrogating the distributed set A for each x using the *lookup* operation discussed in Chapter 2. to see whether l_x is q-awake or not. This requires $O(\lceil |A|/n \rceil \log n)$ time. Step (ii) can be completed in time proportional to the height of the tree. Steps (iii) and (iv) involve updating the distributed sets H and A, and can both be completed in at most $O(\lceil |H|/n \rceil \log n)$ time, using the *update* operation.

It is easy to verify by induction that if any U-node x at pseudoheight h' is marked asleep then the associated vertex l_x in H is not (k,q)-active in $Cull_{k,q}^{(h')}(H)$. These remarks are summarized by the following lemma.

Lemma 14 *If H_{init} and H_{final} represent the state of parameter H before and after the execution of* TreeCull$(\mathcal{F}, H, A, k, q)$, *then H_{final} contains the subgraph* $\mathrm{Cull}_{k,q}^{(h)}(H_{init})$, *where h is the height of the trees in the forest $\mathcal{F}$. Moreover, if the graph H_{init} contains at most $O(n)$ edges then* TreeCull *has a running time of $O(\log n)$.*

5.3 Refinement

The procedure *Refine*(H,k,q) takes a variable parameter H representing a graph, and two value parameters k, and q. Let $H_{init} = (U,V;E_{init})$ and $H_{final} = (U,V,E_{final})$ represent the initial state and final state of the parameter H respectively. If each vertex $u \in U$ has degree at least d_{init} with respect to E_{init}, and if there are at most $\sigma(k)n$ vertices awake in H_{init} then H_{final} is a $(d_{init} - k, q)$-subgraph for Z. The implementation of *Refine* is as follows. The values of the parameters I,J, h_0, and h_1 will be specified later when the reasons for the choices will be clearer.

Refine(H,k,q)

$A :=$ the vertices in H with one or more q-congested edges;

SimpleCull(H,A,k,q,h_0);

for $i := 1$ **to** I **do**

begin

DoLabel$(\mathcal{F}_i, H, A, h_i)$;

for $j := 1$ **to** J **do**

TreeCull$(\mathcal{F}_i, H, A, k, q)$

end

The procedure *DoLabel*$(\mathcal{F}, H, A, h)$ allocates a (u, H)-labelled $(d, \Delta(H), h)$-supertree Θ_u to each $u \in A$, and returns the forest $\{\Theta_u : u \in A\}$ in $\mathcal{F}$. The next section presents an implementation of the procedure *DoLabel* with the running time characterized by the following lemma. (Let $\delta \triangleq \lceil \log d \rceil + \lceil \log \Delta(H) \rceil$.)

Lemma 15 *The sequence of calls to* DoLabel *can be completed in* $O(\log n)$ *time per call provided that (i) the subgraph* H *contains at most* $O(n)$ *edges, (ii)* $|A_i| \leq n/2^{h_i\delta+1}$, *and (iii)* $h_i \leq 2^{i-1}$.

The symbols H_i, and A_i will represent the states of the variables H and A respectively at different points in the computation. Let H_i represent the state of H before the start of the i^{th} iteration of the outer loop. Let A_0 represent the initial value of A, and let A_i represent the set of vertices with one or more q-congested edges in H_i. Let $\rho \triangleq (2/\lambda q)$. The size of the A_i are related as follows. Since the call to *SimpleCull* performs h_0 (k, q)-cullings, it follows that $|A_1| \leq \rho^{h_0-1}|A_0|$ by Lemma 9. Similarly $|A_{i+1}| \leq \rho^{Jh_i}|A_i|$ since the i^{th} iteration performs Jh_i (k, q)-cullings. Setting h_0 to $(\delta+1)/\log\rho^{-1} + 1$ ensures that the number of variables awake just before the first call to *TreeCull* is at most $n/2^{\delta+1}$. Choosing $h_i = 2^{i-1}$ and $J = 3\delta/\log\rho^{-1}$ ensures that

$$|A_i| \leq n\rho^{J(\sum_{l=1}^{i-1} h_l)} \leq \frac{n}{2^{3(h_i-1)\delta}} \leq \frac{n}{2^{h_i\delta+1}},$$

provided that $h_i \geq 2$. Thus the conditions of Lemma 15 are met.

Recall that at most k edges per variable are discarded in the course of the sequence of (k, q)-cullings executed by the calls to *SimpleCull* and *TreeCull*. A total of $s \triangleq h_0 + J\sum_{i=1}^{I} h_i = O(h_0 + J2^I)$ cullings are performed in all. The number of vertices remaining awake after the last call to *TreeCull* is at most $\rho^{s-1}n$. Now $\rho^{h_0+J2^I}n < 1$ for

$$I \geq \log\left\{\frac{\log n}{\log \rho^{-1}} - h_0 - 1\right\} - \log J,$$

so select $I = \log\log n$.

Lemma 16 *If H_{init} and H_{final} represent the state of subgraph H, before and after the execution of* Refine(H, k, q) *respectively, and if H_{init} is a (d_{init}, q_{init})-subgraph of G, then H_{final} is a $(d_{init} - k, q)$-subgraph of H_{init}.*

The overall running time of the procedure *Refine* is as follows.

Lemma 17 *The procedure* Refine(H, k, q), *can be completed in*

$$O\left(\frac{\log \Delta(H) \log n \log\log n}{\log q}\right)$$

time provided that H_{init} contains at most $O(n)$ edges.

Proof To construct the set A from the subgraph H requires $O(\lceil |H|/n \rceil \log n) = O(d \log n)$ time since $|H| = O(n)$ by assumption. The first call to *SimpleCull* runs in

$$O(\lceil H/n \rceil \log n + h_0 \log n) = O(h_0 \log n)$$

time by Lemma 13. Lemma 15 implies that the sequence of *DoLabels* runs in time $O(I \log n)$ time. Each call to *TreeCull* is completed in $O(\log n)$ time. Taken together all these steps can be completed in

$$O\left((h_0 + IJ) \log n\right) \tag{5.2}$$

time. Recall that $h_0 = (\delta + 1)/\log \rho^{-1} + 1$, $I = \log\log n$, $J = 3\delta/\log\rho^{-1}$, and $\delta = O(\Delta(H))$. Substituting these expressions into the above shows that the running time simplifies to

$$O\left(\frac{\log \Delta(H) \log n \log\log n}{\log q}\right).$$

Thus concludes the proof. □

5.4 Labelling the Trees

Now consider the problem of labelling the trees.

To assist in the labelling of the trees the following functions are defined. (It is assumed that the sets $\mathcal{F}$ and H represent a set of labelled trees, and the edges of a subgraph respectively.)

- *tree*$(u, \mathcal{F})$ returns the name of the tree in $\mathcal{F}$, whose root is labelled u.
- *label*$(x, \Theta, \mathcal{F})$ returns the label of the node at position x in the tree Θ in the forest $\mathcal{F}$. (The nodes of a binary tree are numbered in breadth-first order, starting with the root at 1. Thus the children of a node numbered i are numbered $2i$ and $2i+1$. The position of a node x in a tree Θ, is its position with respect to the root of Θ in the above ordering.)
- *edge*(w, l, H) returns the l^{th} edge in the subgraph H incident on vertex w. The edges are numbered in lexicographic order.

All of these can be implemented easily by employing the parallel dictionary techniques of Chapter 2. Any n-tuple of calls to *tree* can be completed in $O(\lceil|\mathcal{F}|/n\rceil \log n)$ time. Similarly any n-tuple of *labels* or n-tuple of *edges* can be completed in $O(\lceil|\mathcal{F}|/n\rceil \log n)$, and $O(\lceil|H|/n\rceil \log n)$ time respectively.

The procedure *DoLabel* consists of two steps as follows:

DoLabel$(\mathcal{F}, H, A, h)$

(1) Allocate a subtree Θ_u of pseudoheight h to each element in A. Label the root of each Θ_u with the name of the corresponding vertex u. Let $\mathcal{F}$ be the collection of trees.

(2) Label the non-root nodes of the trees.

Step (1) can easily be accomplished in $O(\log n)$ time provided that $|A| \leq n/2^{h\delta+1}$. The remainder of this section describes how Step 2 may be completed within the same time bound.

A forest $\mathcal{F}$ of trees of pseudoheight 1 can be labelled as follows.

(1) Let the root r broadcast its label l_r to all of its V-offspring.

(2) Let the i^{th} V-offspring x_i execute $edge(l_x, i, H)$ to determine the i^{th} edge (l_x, v_i) in H incident on l_x. Label x_i with v_i.

(3) Let each V-node x broadcast its label l_x to all of its U-offspring.

(4) Let j^{th} U-offspring of x execute $edge(l_x, j, H)$ to determine j^{th} edge (l_x, u_j) in H incident on l_x. Label node x with u_i.

The above involves two calls to the procedure *edge*, and can be completed in $O(\lceil |H|/n \rceil \log n)$ time.

Trees of pseudoheight greater than 1 are labelled by *pasting* together the labellings of the trees from the previous iteration of the outer loop of *Refine*. The key observation is the following. Let Θ_u be an (u, H)-labelled tree of pseudoheight h, and suppose x is some U-node in Θ_u labelled u' at pseudoheight $h' \leq h$; then the subtree rooted at x is (u', H)-labelled.

Let $\mathcal{F}_\mathrm{i}$ denote the set of labelled trees constructed during the i^{th} iteration. Let $\Theta_i(u)$ denote the tree in $\mathcal{F}_\mathrm{i}$ whose root is labelled u. Recall that $h_{i+1} = 2h_i$.

To label a tree $\Theta_{i+1}(u)$ in $\mathcal{F}_{\mathrm{i+1}}$ first label the top h_i levels of the tree by copying the labelling from $\Theta_i(u)$. For each node x in each $\Theta_{i+1}(u)$ at pseudoheight h_i, check if l_x is awake and if so label the subtree of $\Theta_{i+1}(u)$ rooted at x by copying the labelling of $\Theta_i(l_x)$. To copy the labelling of a tree Θ to Θ', each node in Θ' at a position x, reads the label of the corresponding node at position x in Θ, by executing a call to *label*. The details are as follows.

(1) For each $a \in A$ execute $tree(a, \mathcal{F}_i)$ to determine the tree in $\mathcal{F}_i$ with root labelled a. Broadcast the name $\Theta_i(u)$ to all nodes in the tree $\Theta_{i+1}(u)$.

(2) For each Θ_{i+1} and for each U-node and V-node x at pseudoheight h_i or greater, let $\pi(x)$ denote its position within $\Theta_{i+1}(u)$ with respect to the root. For each such node x execute a *label*($\pi(x), \Theta_i(u), \mathcal{F}_i$) to determine the label for node x.

(3) For each $x \in \mathcal{F}_{i+1}$ at pseudoheight h_i first check if l_x is in A, and if so execute a *tree*($l_x, \mathcal{F}_i$) to determine $\Theta_i(l_x)$. Let each such node x broadcast $\Theta_i(l_x)$ to all of its descendants in the tree $\Theta_{i+1}(u)$.

(4) For each node y at position z in the subtree rooted at x execute a *label*($z, \Theta_i(l_x), \mathcal{F}_i$) call and take the label returned as the label of node y.

The entire labelling process can be completed in completed in $O(\lceil(|A| + |H|)/n\rceil \log n)$ time. This establishes Lemma 15

5.5 Constructing Subgraphs

The overall construction of the the subgraph H_Z may now be specified.

Step II

$H := G(Z)$;

$A :=$ vertices of H with one or more q-congested edges;

SimpleCull($H, A, \lceil d/2\rceil - 1, q_G, t$);

$H_1 :=$ subgraph induced by vertices with no q-congested edges;

$H_2 :=$ subgraph induced by q-awake vertices;

for $l := 1$ **to** L **do**

Refine($H_2, d/r, q_l$);

$H_Z := H_1 \cup H_2$;

By Lemma 9 after the call to *SimpleCull* the size of A is at most $(2/\lambda q_G)^{t-1}n$ which is at most $\sigma(d/r)n$ by setting

$$t = \frac{\log \sigma^{-1}(d/r)}{\log(\lambda q_G/2)} + 1 = O(r),$$

since $q_G = (2/\lambda)dg^2$ by definition.

Following this first call to *SimpleCull*, H_1 is a $(\lfloor d/2 \rfloor + 1, q_G)$-subgraph for $Z' \subseteq Z$ which contains all all but at most $\sigma(d/r)n$ variables in Z, and H_2 is the subgraph of G induced by $Z'' = Z - Z''$. The bound on the size of Z'' implies that the techniques of Subsection 5.3 may be applied to refine H_2. Each refinement can discard at most d/r edges belonging to each variable, but provided $L(d/r) < d/2$ *i.e* $r > 2L$ and $q_L \leq q_G$, then H_2 is a $(\lfloor d/2 \rfloor + 1, q_G)$-subgraph for Z'' after the last refinement.

The call to SimpleCull requires

$$O(r \log n + d \log n)$$

time. The l^{th} call to *Refine* runs in $O((\log q_{l-1}/\log q_l) \log n \log\log n)$ time by Lemma 17 ($q_0 \triangleq \Delta(G)$). Therefore the total cost for Step II is

$$O\left(d \log n + r \log n + \log n \log\log n \sum_{l=1}^{L} \frac{\log q_{l-1}}{\log q_l}\right).$$

By choosing $q_l \triangleq (\Delta(G))^{1/2^l}$ for $l = 0, 1, \ldots, L$, it can be seen that setting the parameter $L = O(\log\log(m/n) - \log\log q_G)$ refinements suffice to reduce the congestion of the subgraph H_2 to $O(q_G)$ since $\Delta(G) = O((m/n)d)$. Moreover each summand in the sum is $O(1)$ in this case. Choosing $r = 2L + 1$ and simplifying, the expression for the running time becomes

$$O\left(d \log n + \log n \log\log n(\log\log(m/n) - \log\log\log n)\right).$$

As observed earlier the combined cost of all the other steps in the simulation is $O(d \log n)$ and this is subsumed by the above expression.

Thus the entire simulation can be completed in

$$\mathcal{S}_{BDN}(m,n) \triangleq O\left(d\log n + \log n \log\log n(\log\log(m/n) - \log\log\log n)\right)$$

time which is

$$O\left(\log(m/n)\log n/\log\log n\right.$$
$$\left.+\log n\log\log n(\log\log(m/n) - \log\log\log n)\right).$$

For ease of reference, observe that the construction was in two parts the first consisting of $O(r)$ culling steps effected by a call to *SimpleCull* and the second part involving a sequence of $r = \log\log(m/p) - \log\log q_G$ refinements.

5.6 Simulations on Larger Networks

In this section we extend the result established in the previous section to the case where p the number of nodes in the simulating BDN is greater than n the number of PRAM processors. We will show that faster simulations are possible in some cases.

One of the chief limitations on the speed of simulations is the time required to route information around the network. In Chapter 2 it was established that the time required to route any instance of the (N, K_1, K_2) routing problem is $O(\lceil N/p\rceil \log p + K_1 + K_2)$ time on a particular p-node BDN. This suggests that larger networks can route large amounts of information faster than smaller ones since the workload can be shared among a larger number of processors. This observation can be exploited to provide simulations faster than those of the previous section in some cases. The modifications to the algorithm are relatively minor.

Let

$$g \triangleq 2\left(\sqrt{(n/p)\sqrt{\log p}} + 1\right),$$

and let the memory organization be a balanced $(m, p, \lambda. d, d/r, \sigma)$-generalized expander with $d = O(\log(m/p)/\log g)$, $q_G = (2/\lambda)dg^2 + \log p$, λ a constant less than one, $r = O(\log\log(m/p) - \log\log q_G)$, and $\sigma(k) = (n/p)/dg^{d/k}$.

The algorithm has exactly the same structure as that described in the earlier sections of this chapter except for the choice of some parameters.

To access a set Z of variables, where $|Z| \leq n$, the algorithm first constructs an $(\lfloor d/2 \rfloor + 1, q_G)$-sheaf F_Z for Z, and then proceeds to access the copies corresponding to the edges in F_Z. By the remarks made above the copies associated with F_Z can be accessed in $O(\lceil (n/p)d \rceil \log p + d + q_G)$ time.

The running time of the algorithm may be reanalysed as follows. The dominant cost of the call to *SimpleCull* during Step II is that of the routing steps involved. When recalculated in terms of a p-node BDN this substep has a running time of

$$O\left(\lceil (n/p)d \rceil \log p + r \log p + d\right).$$

The l^{th} refinement reduces the congestion of the subgraph H from $\Delta(G)^{1/2^{l-1}}$ to $\Delta(G)^{1/2^l}$. Since $\Delta(G) = O((m/p)d)$ in is easy to see that $L = O(\log\log(m/p) - \log\log\log q_G)$ refinements suffice to reduce the congestion to q_G. The running time of the whole sequence of calls to *Refine* is

$$O\left(L \log p \log\log n\right)$$

Summing the cost of the call to *SimpleCull*, the sequence of calls to *Refine*, and the time required to access copies associated with the sheaf F_S the whole simulation can be completed in time

$$O(d + q_G + \lceil (n/p)d \rceil \log p + L \log n \log\log n)$$

which simplifies to

$$O\left(\lceil (n/p)d \rceil \log p + d \left\lceil (n/p)\sqrt{\log p} \right\rceil \right.$$
$$\left. + \log p \log\log n (\log\log(m/p) - \log\log\log n)\right)$$

where

$$d = O\left(\log(m/p) \Big/ \log\left\lceil (n/p)\sqrt{\log p}\right\rceil\right).$$

Chapter 6

Simulations on an Augmented Mesh-of-Trees Architecture

The simulation presented in Chapter 5 is modified here to run on an augmented mesh-of-trees (AMOT). An n-processor AMOT has $O(n^2)$ switching nodes and in many cases can simulate a PRAM faster than a n-processor BDN.

6.1 The Mesh-of-Trees

The $n \times n$ two-dimensional *mesh-of-trees* (MOT) architecture [Lei81,NMB83] consists of some number n processor nodes $P_1, P_2, \ldots, P_n$, as well as n^2 leaf nodes $\{l_{i,j} : 1 \leq i \leq n \, and \, 1 \leq j \leq n\}$, and two sets of binary trees $\{C_1, C_2, \ldots, C_n\}$ (*column trees*), and $\{R_1, R_2, \ldots, R_n\}$ (*row trees*). The row tree R_i is a complete binary tree rooted at processor node P_i that has the nodes in $\{l_{i,j} : 1 \leq j \leq n\}$ as leaves. The column tree C_j is a complete binary tree rooted at processor node P_j that has the leaf nodes in $\{l_{i,j} : 1 \leq i \leq n\}$ as leaves. The row tree R_i and the column tree C_j have a single leaf node $l_{i,j}$ in common.

The simplicity and regularity of the mesh-of-trees make it ideally suited to VLSI implementation. Parallel algorithms for a variety of problems such as sorting, DFT *etc.* [NMB83] have been devised for this architecture.

The MOT has also been proposed as a general-purpose parallel architecture. Luccio *et al.* [LPP87,LPP88] consider the problem of simulating an (n,m)-PRAM on an n-node MOT. Under the assumption that m is polynomial in n, they obtain a randomized simulation [LPP88] with an expected running time of $O(\log n)$. In the deterministic case, they present [LPP87] a simulation with a worst-case running time of $O(\log^2 n/\log\log n)$. The memory organization employed by their deterministic simulation is the same as the one presented in [UW87], and involves $O(\log m)$ copies of each variable. Hornick and Preparata [HP89] tailored the memory organization to a $n^{1+\epsilon} \times n^{1+\epsilon}$ mesh-of-trees and obtained a simulation with the same running time as that quoted above using $O(1)$ copies per variable, when $m = \Omega(n^{2+\epsilon})$.

The $n \times n$ augmented mesh-of-trees (AMOT) has the structure of an $n \times n$ mesh-of-trees but with additional links that join the processor nodes in the structure of a complete binary tree of n nodes. Each of the channels in the row and column trees as well as those in the tree joining the processors is an $O(\log m)$ bit wide connection linking its endpoints. It is assumed that a single $O(\log m)$ bit quantity may be transmitted along any edge in $O(1)$ time. The processor nodes are standard RAM processors [AHU74]. The leaf nodes and the internal nodes of the row and column trees are simple switches and can buffer $O(1)$ messages. To assist in sorting it is further assumed that each of the leaves contains a comparator and that each of the nodes in the row and column trees contains an adder.

This chapter presents a simulation of an (n,m)-PRAM that runs in

$$O(\log(m/n) + \log n \log\log n(\log\log(m/n) - \log\log\log n)).$$

time on a $n \times n$ AMOT. Notice that this is faster than that claimed in Theorem 3 in some cases. As we will see in Chapter 7 achieving the space bounds stipulated in Theorem 3 results in a slightly slower algorithm.

It should be observed that the AMOT with n processors has $\Theta(n^2)$ communication channels compared with $\Theta(n)$ for a n-node BDN. It turns out that this can be exploited to provide very efficient generalized routing and sorting algorithms on the AMOT. These algorithms are faster than the best algorithms for these problems on a BDN with the same number of processors in some cases.

6.2 Communication Algorithms

The simulations seen so far rely on the following communication primitives: (i) sorting, (ii) partial permutation routing, (iii) generalized routing and balancing, and (iv) tree-based computations such as prefix and TreeCull.

The tree connections joining the processors of a $n \times n$ MOT imply that the tree-based computations such as prefix can be accomplished in the same running time as on the n-node BDN of the previous chapter. In fact these operations could also be executed on one of the row or column trees on an (unaugmented) MOT. This approach is not adopted since this would add to the complexity of the internal nodes of these trees.

It is possible to sort n items on a n-node MOT in $O(\log n)$ time [NMB83]. In fact this sorting algorithm can be pipelined so that s sets of n items each may be sorted in $O(s + \log n)$ time.

Consider the problem of routing a set $\hat{Z} = (Z, f_1)$ of packets on an MOT. (Recall that $f_1(x)$ indicates the node holding packet x.) More precisely, suppose that we wish to redistribute the packets according to f_2. For $i = 1, 2$, let f_i have profile $(k_1^i, \cdots, k_n^i)$ and degree Δ_i.

Assign each packet x a *starting time* st_x so that no two packets with the same starting time have the same origin or the same destination.

In fact for any f_1 and f_2 there exists an assignment of starting times to packets such that no packet has a starting time of more than $2\max\{\Delta_1, \Delta_2\}$. This may be seen as follows. Assume that no two packets with the same origin have the same destination. Consider the bipartite graph $B = (U, V; E)$ where $|U| = |V| = n$ and $(u, v) \in E$ represents the fact that there is a packet with origin u and destination v. Each edge in B is incident upon at most $2\max\{\Delta_1, \Delta_2\} - 2$ others. Colour the edges of E in some order; each edge receives the lowest numbered colour not already present among the colours of the adjacent edges. At most $2\max\{\Delta_1, \Delta_2\} - 1$ colours suffice. Each colour represents a different starting time. The above restriction on the destinations of the packets with a common origin can be removed by considering multiple edges.

In our applications it will either be the case that f_1 and f_2 are fixed (in which case the starting times may be computed off-line), or else f_1 and f_2 are such that it is relatively easy to compute the appropriate starting times on-line.

Routing the set of packets with a given starting time is a partial permutation problem, which can be solved in $O(\log n)$ time: each packet originating at a with destination b simply follows the edges of the column tree C_a to leaf $l_{a,b}$ and then uses row tree R_b to advance towards its destination. It is easy to see that each packet can advance across one link in each time step without interfering with the progress of other packets. It is also easy to see that a number w of partial-permutations can be pipelined in $O(w + \log n)$ time. Thus (Z, f_1) can be transformed to (Z, f_2) in $O(\Delta_1 + \Delta_2 + \log n)$ time (assuming that the starting times are known).

Recall that the generalized routing algorithm developed in Chapter 2 rested on the concept of balancing. Similarly, the (N, K_1, K_2)-routing problem can be solved in $O(\log n + K_1 + K_2)$ time on a $n \times n$ AMOT, given the algorithm to approximately balance a distributed set $\hat{Z}$ of packets $O(\log n + \Delta(\hat{Z}))$ time. Let $k = \lceil |\hat{Z}|/n \rceil$.

A distributed set $\hat{Z}$ may be balanced by choosing for each packet x both a destination g_x and a starting time st_x as follows. First number the packet in increasing order of origin (packets with the same origin are numbered consecutively). Let N_x be the number of packet x in this ordering. Let

$$\begin{aligned} g_x &= N_x \, div \, k + 1, \text{ and} \\ st_x &= N_x \, mod\,(\Delta(\hat{Z}) + k + 1). \end{aligned}$$

The numbering N_x and the value k can each be computed in $O(\log n + \Delta(\hat{Z}))$ time by parallel prefix techniques. It is easy to see that the packets with the same starting times form a partial permutation problem. By the definition of st_x no two packets with the same origin can have the same starting time. Suppose that two packets x and y have the same starting time and the same destination. Since $g_x = g_y$ we know that $N_x + 1 \leq N_y \leq N_x + k - 1$ (assuming that $N_x \leq N_y$ without loss of generality). This implies that $N_x \bmod P$ can not equal $N_y \bmod P$, where $P \triangleq \Delta(\hat{Z}) + k + 1$. Since there are $\Delta(\hat{Z}) + k + 1 = O(\Delta(\hat{Z}))$ different starting times the entire routing can be completed in $O(\Delta(\hat{Z}) + \log n)$ time.

Leighton's columnsort [Lei85] may be summarized as follows. Let the array $A[1..r, 1..s]$ be an $r \times s$ array of values with $r \geq 2(s-1)^2$. The array A may be sorted (in column major order) in eight steps where the odd numbered steps involve sorting the s columns of A in parallel and the even numbered steps involve permuting the elements of A. The permutations applied depend only on r and s.

The balanced distributed set $\hat{Z}$ may be sorted as follows. Partition $\hat{Z}$ into $k = \lceil |\hat{Z}|/n \rceil$ *strata* $\hat{Z}_1, \ldots, \hat{Z}_k$ of size at most n each. By the above reasoning any permutation of the elements of $\hat{Z}$ may be completed in $O(k + \log n)$ time on the MOT. (Since the permutations are fixed the appropriate starting times can be computed off-line.) Furthermore each of the strata can be sorted in pipelined fashion in $O(k + \log n)$ time, and so it follows that any distributed set can be sorted in

$$O\left(\left\lceil\frac{|\hat{Z}|}{n}\right\rceil + \log n\right)$$

time on a $n \times n$ MOT provided that $|\hat{Z}| \leq n(\sqrt{n/2})+1$. No doubt this result can be generalized to sets of larger size but we will not pursue this question here. It should be noted that sorting a set of size N requires at least $\Omega(\lceil N/n\rceil \log n)$ time on any n-node BDN.

Thus, each of the communication primitives listed at the start of this section may be implemented at least as efficiently on a $n \times n$ AMOT as the n-node BDN described in Chapter 5. Indeed all except the tree based computation can be implemented on the (unaugmented) mesh-of-trees.

6.3 Simulations on the Augmented MOT

In this section the simulation presented in Chapters 5 is modified to run on an n-processor AMOT. The memory organization will be a $(m, n, \lambda, d, d/r, \sigma)$-generalized expander $(U, V; E)$ where λ is a constant less than one, $r = \log\log(m/n) - \log\log\log n$, and $\sigma(k) = 1/d2^{d/k}$. By Theorem 3, this memory organization has

$$d = O(\log(m/n))$$

copies per variable. Notice that in general this memory organization has more copies than the memory organization presented in Chapter 5 for the n-node BDN. For the moment it is assumed that the representation of the memory organization is replicated in the local memories of the processors. A more space efficient algorithm will be presented in the next chapter.

Recall from Chapter 5 that the algorithm for accessing some n-tuple Z of variables has the following high level structure. Let $q_G \triangleq (8/\lambda)d + \log n$.

1. Construct a $(\lfloor d/2 \rfloor + 1, q_G)$ sheaf $F_Z = F_{Z'} \cup F_{Z''}$ for Z:

   (a) Construct a $(\lfloor d/2 \rfloor + 1, q_G)$ sheaf $F_{Z'}$ for $Z' \subseteq Z$ containing all but a few variables of Z;

   (b) Construct a $(\lfloor d/2 \rfloor + 1, q_G)$ sheaf $F_{Z''}$ for $Z'' = Z - Z'$:

2. Physically access the copies represented by F_Z and route them to the intended processors.

Recall from Section 5.6 that Step 1(a) was based on the procedure *SimpleCull* that performed $O(r)$ culling steps. The dominant cost of which was that of sorting a balanced distributed set of size nd. Substituting the sorting algorithm developed in the previous section this can be accomplished in $O(d + \log n + r)$ time, provided that $nd = O(n^{3/2})$.

The sheaf construction of Step 1(b) involved r refinements and the implementation was based on the communication primitives listed at the start of the previous section. By the results of Section 6.2 any algorithm running in time T on an n-node BDN based entirely on these primitives can also be completed in $O(T)$ time on an n-processor AMOT. In Chapter 5 we saw that each refinement required in $O(\log n \log\log n)$ time on an n-node BDN and so this the r refinements can be completed in

$$O(\log n \log\log n(\log\log(m/n) - \log\log\log n))$$

time on the $n \times n$ AMOT.

Finally, Step 2 involves an instance of the (N, K_1, K_2)-routing problem with $N = nd$, $K_1 = d$, and $K_2 = q_G = O(d)$. By the results of Section 6.2, this can be completed in

$$O(d + \log n)$$

time. Summing the contributions of the various terms we see that the simulation can be completed in

$$O(d + \log n \log\log n(\log\log(m/n) - \log\log\log n))$$

time in all.

We have established the following

Lemma 18 *For every m, and n with $n \leq m \leq n2^{\sqrt{n/2}+1}$ there is a deterministic algorithm that can simulate any step of an (n, m)-PRAM in*

$$O(\log(m/n) + \log n \log\log n(\log\log(m/n) - \log\log\log n)) \tag{6.1}$$

time on an $n \times n$ AMOT.

In the next chapter we will see how the space requirements for the algorithm can be reduced to the bounds stated in Theorem 2.

Chapter 7

Space-Efficient Simulations

In this chapter we show how the space requirements for the simulations of the previous chapters can be reduced to bounds stipulated in Theorems 1 and 2. We focus first on the BDN presented in Chapter 5, and then extend the results to the AMOT.

7.1 Introduction

In Chapter 5 we showed that if m variables are distributed among the n nodes of a BDN according to a generalized expander $G = (U, V; E)$, then for any n-tuple or variables, a quorum of copies for each variable can be accessed in $O(\mathcal{S}_{BDN}(m, n))$ time. The processors must be able to query the structure of G in order to determine the locations of the copies. The approach taken in previous studies (and in the previous chapters) is to store in the local memory of each node a table listing for each vertex $u \in U$ the edges in E incident upon it. As we have already pointed out this is extremely wasteful of storage.

The goal of this chapter is to develop a more compact representation of the memory map such that the locations of the copies of any n-tuple of variables in G can be determined in $O(\mathcal{S}_{BDN}(m.n))$ time.

Abstractly, the structure of G may be represented by a m-tuple of *values*

$\mathbf{x}_G = (x_1, x_2, \ldots, x_m)$, where each value x_i consists of d *items* (the d edges incident on the i^{th} vertex in G). We say that $\mathbf{x}_G$ has length m and width d. Since the set of edges belonging to a particular vertex will always be read as unit the structure of G will be represented as an m-tuple of width d rather than an (md)-tuple of width 1.

Considerations of the type that lead to the development of generalized expanders suggest that the values of $\mathbf{x}_G$ will have to be replicated and the copies distributed with care to control congestion when accessing $\mathbf{x}_G$. Since the address map will never be altered it can be read-only; hence the algorithm need only read one copy per item. Thus the work involved in routing d items for each of n values is $O(d \log n)$.

A *distributed array* $(\mathbf{x}, \varphi)$ is a pair where $\mathbf{x}$ is an m-tuple $(x_1, \ldots, x_m)$ of values and $f : [0..m-1] \rightarrow 2^{[0..n-1]}$ is a function that describes the distribution of copies of the values to nodes in the network: each of the nodes whose index lies in the set $f(i)$ contains a copy of the value x_i.

The main result of this chapter is the following.

Theorem 7 *Let m, and n be any integers satisfying $n \leq m \leq n2^{2^{\sqrt{\log n}}}$, and let d denote the quantity $\log(m/n)/\log\log n$. There is a deterministic algorithm running on (V, E_{route}) such that any m-tuple $\mathbf{x}$ of width at most d may be represented using $O((m/n)d^2(\log\log(m/n) - \log\log\log n))$ storage per node, and any n-tuple of values from $\mathbf{x}$ can be probed in $O(S_{BDN}(m, n))$ time.*

Consider a distributed array $(\mathbf{x}, \varphi)$ of width at most d. Let us assume that there are c copies per value. The task of accessing an n-tuple S of values from $(\mathbf{x}, \varphi)$ involves (i) evaluating φ at various points to determine the location of the copies of S, (ii) selecting a subset of $c' \leq c$ copies of each value in S, and (iii) accessing $\lceil d/c' \rceil$ items for each value from each selected node.

The choice of φ is made with the following two constraints in mind. The first is that φ should distribute the values so that they are accessible *i.e.* Steps

(ii) and (iii) can be completed efficiently. The second consideration is that the representation of φ should be economical with respect to storage, and still admit efficient evaluation.

The approach taken is to construct φ from functions derived from certain generalized expanders. Intuitively, the idea is that by building φ from components with good expansion characteristics, the distribution defined by φ will ensure that the copies of each set of values are spread over as many nodes as possible. Furthermore, by building φ from "smaller" pieces, these pieces can be represented by distributed arrays each of length much less than m. Thus the problem of representing the distribution function φ reduces to that of representing a number of distributed arrays of length much smaller than m. In all the algorithm will maintain a number of distributed arrays in addition to $(\mathbf{x}, \varphi)$ itself; the role of $(\mathbf{x}, \varphi)$ is to represent the m-tuple $\mathbf{x}$; the role of the other distributed arrays is to represent φ.

7.2 Distributing the Vector

In this section functions φ_i, for $i = 0, 1, 2, 3$ are defined that describe the distributions of copies for the various distributed arrays employed by the algorithm presented in the next section.

We begin by introducing a correspondence between functions and bipartite graphs.

Definition 6 *For any function $f : [0..m-1] \rightarrow 2^{[0..n-1]}$ be a function, let $G(f) = (U, V; E)$ denote a graph with $U = \{u_0, \ldots, u_{m-1}\}$, $V = \{v_0, \ldots, v_{n-1}\}$, and $E = \{(u_i, v_j) : j \in f(i)\}$.*

The definitions of the functions φ_i are fairly involved. First we define certain generalized expanders and then describe the φ_i in terms of the functions associated with these expanders.

Let $m_k = m/(m/n)^{k/3}$, and $n_k = n/(m/n)^{k/3}$. Also let d denote the quantity $\log(m/n)/\log\log n$, and let r represent the value $\log\log(m/n) - \log\log\log n$.

Definition 7 *let $G_k^{(1)}$ be a balanced*

$$(m_{k+1}, m_{k+2}, 1/2, r, a, \sigma_1) - \textit{generalized expander},$$

where

$$\sigma_1(h) = \frac{n_k/m_{k+2}}{r2^{r/h}},$$

and a is a constant independent of m, and n. Let $G_k^{(2)}$ be a balanced

$$(m_{k+1}, n_k, 1/2, d, d/r, \sigma_2) - \textit{generalized expander},$$

where

$$\sigma_2(h) = \frac{1}{d(\log n)^{d/4h}}.$$

By Theorem 5 both $G_k^{(1)}$ and $G_k^{(2)}$ exist. Let $g_k^{(1)}$ and $g_k^{(2)}$ be the functions defined by $G(g_k^{(1)}) = G_k^{(1)}$, and $G(g_k^{(2)}) = G_k^{(2)}$ respectively.

The function $\varphi_k : [0..m_k - 1] \rightarrow 2^{[0..n-1]}$ will be defined in terms of $g_k^{(1)}$ and $g_k^{(2)}$. Intuitively, φ_k will be defined as the composition of two functions ψ_k and θ_k. The function ψ_k will be defined by "pasting" together several copies of the function $g_k^{(1)}$ in a sense made precise below. The function θ_k is obtained in a similar way from $g_k^{(2)}$.

Let $[a..b]$ denote the integer interval $\{a, a+1, \ldots, b-1, b\}$. Let

$$\begin{aligned}
A^{(k)} &\triangleq [0..m_k - 1], \\
A_l^{(k)} &\triangleq [lm_{k+1}..(l+1)m_k - 1], \\
B_l^{(k)} &\triangleq [lm_{k+1}..(l+1)m_{k+1} - 1], \\
B_{l,h}^{(k)} &\triangleq [lm_{k+1} + hm_{k+2}..lm_{k+1} + (h+1)m_{k+2} - 1], \\
C_l^{(k)} &\triangleq [ln_k..(l+1)n_k - 1]
\end{aligned}$$

Notice that $A^{(k)} = \cup A_l^{(k)}$ and that $B_l^{(k)} = \cup B_{l,h}^{(k)}$. Let $f : A \to 2^B$ be a function, and let $X \subseteq A$, and $Y \subseteq B$ be subsets of A and B respectively. Let $f[X, Y]$ be the function defined by

$$f[X, Y] : X \to 2^Y : x \mapsto Y \cap f(x).$$

Let $N_k = (m/n)^{k/3}$. The function θ_k is defined in such a way that for each l in the range $0 \leq l \leq N_k - 1$ the graph $G(\theta_k[B_l^{(k)}, C_l^{(k)}])$ is isomorphic to $G_k^{(2)}$. Thus, the graph $G(\theta_k)$ is isomorphic to the graph consisting of N_k disjoint copies of the graph $G_k^{(2)}$.

Definition 8 *Let* $\theta_k : [0..m_1 - 1] \to 2^{[0..n_k-1]}$ *be defined by*

$$\begin{aligned}\theta_k(x) \quad = \quad & \{y = (x \operatorname{div} m_{k+1})n_k + z, \\ & \qquad \textit{where } z \in g_k^{(2)}(x \bmod m_{k+1})\}.\end{aligned}$$

Similarly, the function ψ_k is defined so that that for each i and j such that $0 \leq i \leq N_k - 1$ and $0 \leq j \leq (m/n)^{1/3}$ the graph $G(\psi_k[A_j^{(k)}, B_{i,j}^{(k)}])$ is isomorphic to $G_k^{(1)}$.

Definition 9 *Let* $\psi_k : [0..m_k - 1] \to 2^{[0...m_1-1]}$ *be defined by*

$$\begin{aligned}\psi_k(x) \quad = \quad & \{y = lm_{k+1} + m_{k+2}(x \operatorname{div} m_{k+1}) + z, \\ & \qquad \textit{for some}\, 0 \leq l \leq N_k - 1 \\ & \qquad\quad \textit{and some}\, z \in g_k^{(1)}(x \bmod m_{k+1})\}.\end{aligned}$$

Definition 10 *For* $k = 0, 1, 2,$ *let*

$$\varphi_k : [0..m_k - 1] \to 2^{[0..n-1]} : x \mapsto \theta_k(\psi_k(x)).$$

Let $\varphi_3 : [0..n - 1] \to 2^{[0..n-1]} : x \mapsto \{x\}$.

The storage regime may now be described. There are seven distributed arrays in all:

$(\mathbf{x}, \varphi_0)$	the top level m-tuple,		
$(\mathbf{x}(g_0^{(1)}), \varphi_1)$	a map for $g_0^{(1)}$,	$(\mathbf{x}(g_0^{(2)}), \varphi_1)$	a map for $g_0^{(2)}$,
$(\mathbf{x}(g_1^{(1)}), \varphi_2)$	a map for $g_1^{(1)}$,	$(\mathbf{x}(g_1^{(2)}), \varphi_2)$	a map for $g_1^{(2)}$,
$(\mathbf{x}(g_2^{(1)}), \varphi_3)$	a map for $g_2^{(1)}$, and	$(\mathbf{x}(g_2^{(2)}), \varphi_3)$	a map for $g_2^{(2)}$.

Each of these distributed arrays has width at most d.

Note that the distribution φ_0 of the top level distributed array $(\mathbf{x}, \varphi_0)$, depends on the functions $g_0^{(1)}$, and $g_0^{(2)}$ which are represented by the distributed arrays $(\mathbf{x}(g_0^{(1)}), \varphi_1)$, and $(\mathbf{x}(g_0^{(2)}), \varphi_1)$ respectively. The function φ_1 describing the distribution of values of these arrays depends on $g_1^{(1)}$, and $g_1^{(2)}$, and so on.

The following lemma shows that the storage regime meets the space bounds specified in Theorem 7.

Lemma 19 *All of the distributed arrays specified above can be stored using at most $O((m/n)d^2r)$ storage per node.*

Proof We begin by showing that for $k = 0, 1, 2, 3$ the set of values in the domain mapped by φ_k to any individual node is at most $O((m/n)dr)$. The result is immediate in the case of φ_3. Let $\Delta(f)$ denote the maximum number of values mapped to any point by the function f. Clearly $\Delta(\varphi_k) = \Delta(\psi_k)\Delta(\theta_k)$ for $k = 0, 1, 2$.

It is easy to see that $\Delta(\psi_k) = \Delta(g_k^{(1)})$ which is $O((m_{k+1}/m_{k+2})r) = O((m/n)^{1/3}r)$ since $G(g_k^{(1)})$ is balanced. Similarly, we can show that $\Delta(\theta_k) = O((m/n)^{2/3}d)$. Thus

$$\Delta(\varphi_k) = \Delta(\psi_k)\Delta(\theta_k) = O((m/n)dr),$$

as claimed.

Since each of the distributed arrays has width at most d it follows that at most $O((m/n)d^2r)$ storage per node will be required to store all of the distributed arrays. □

In the next section we will show that any n-tuple of values of $\mathbf{x}$ may be accessed in $O(\mathcal{S}_{BDN}(m,n))$ time.

7.3 Accessing a Distributed Vector

As mentioned in the introduction, there are three distinct tasks involved in accessing an n-tuple of values from a distributed array $(\mathbf{a}, f)$: (i) determining the locations of the copies by evaluating f, (ii) selecting which copies to access, and (iii) physically accessing the desired items. For reasons of efficiency the actual algorithm will not mirror this scheme exactly: the tasks of determining the locations of the copies, and determining which copies to access will be intertwined. Informally the reason is as follows. The degree of ψ_0 is r, and that of θ_0 is d, and so there are dr copies of each value x_i of $\mathbf{x}$. If we were to determine the locations of all dr copies of some n-tuple of variables then routing this information to the intended processors would require $O(dr \log n)$ time, which would exceed our stipulated time bound. Instead, we will determine the locations of a cluster of d copies, and then select $\lfloor d/2 \rfloor + 1$ of these. The selection will be made by first choosing for each x_i a single y_i such that $y_i = \psi_0(x_i)$, then selecting $\lfloor d/2 \rfloor + 1$ nodes $z_i^1, \ldots, z_i^{\lfloor d/2 \rfloor + 1}$, such that $z_i^j \in \theta_0(y_i)$, for $j = 1, \ldots \lfloor d/2 \rfloor + 1$. The selections in both cases is made in a way that guarantees that no node z is chosen as one of the z_i^j for too many x_i. Recall that each value consists of at most d items. The algorithm will read two items for each x_i from each of the nodes z_i^j.

Definition 11 *Let $f : [0..a-1] \rightarrow 2^{[0..b-1]}$ be a function, and let $S \subseteq [0..a-1]$. Then a set X of ordered pairs in $[0..a-1] \times [0..b-1]$ is an (α, β)-set for S with respect to f if (i) each $s \in S$ appears at least α times as the first component of* *a pair in X, (ii) each $t \in [0..b-1]$ appears as a second component no more then β times, and (iii) each pair in X has the form (s,t) where $t \in f(s)$.*

For $i = 0,1,2$, let T_i denote the time required to access any n-tuple of values from a array distributed according to φ_k. Clearly $T_3 = O(\log n)$. Let $q_1 \triangleq (8/\lambda)r$, and let $q_2 \triangleq (2/\lambda)d\sqrt{\log n}$.

The algorithm to read an n-tuple X from a array distributed according to φ_k $(k = 0,1,2)$ is as follows.

(1) Determine $\psi_k(x)$ for each $x \in X$.

(2) Construct a $(\lfloor r/2 \rfloor + 1, q_1)$-set $\tilde{X}$ for X with respect to ψ_k.

(3) For each $x \in X$ select one pair (x,y) from $\tilde{X}$. Let the set of selected pairs be denoted by $\tilde{X}'$. Let Y be the set $\{y : (x,y) \in \tilde{X}'\}$.

(4) Determine $\theta_k(y)$ for each $y \in Y$.

(5) Construct a $(\lfloor d/2 \rfloor + 1, q_2)$-set $\tilde{Y}$ for Y with respect to θ_k.

(6) Let $\mathcal{X}$ be the set $\{(x,z) : x \in X \, s.t. \exists y with (x,y) \in \tilde{X}' \, and (y,z) \in \tilde{Y}\}$.

(7) For each $(x, z_x^i) \in \mathcal{X}$ access two items of value x from node z_x^i.

Notice that for each $x \in X$, the set $\mathcal{X}$ contains at least $\lfloor d/2 \rfloor + 1$ pairs of the form (x, z_x^i), and that for each z_x^i contains a copy of x since $z_x^i \in \varphi_k(x)$.

The running time of this algorithm may be analysed as follows. Step (1) involves probing the distributed array representing the structure of $G_k^{(1)}$ which requires T_{k+1} time. Step (4) requires the same amount of time. Later in this section we will show that Steps (2) and (5) can be accomplished in time $O(\mathcal{S}_{BDN}(m,n))$ each. Step (3) involves straightforward manipulations of a distributed set of $O(nr)$ packets and can be accomplished by a combination of sorting and prefix techniques in $O(r \log n)$ time. Similarly Step (6) can be completed in $O(d \log n)$ time. Step (7) involves routing a distributed set of $O(nd)$ packets, with at most $\lfloor d/2 \rfloor + 1$ per source and at most $q_1 q_2$ per destination. This can be accomplished in time

$$O(d \log n + q_1 q_2) = O(d\sqrt{\log n}(r + \sqrt{\log n}))$$

which is $O(d \log n)$ for $m \leq n2^{2^{\sqrt{\log n}}}$. Thus the running time T_k is at most

$$O(d \log n + \mathcal{S}_{BDN}(m, n) + T_{k+1}),$$

under the above mentioned constraint on m. Since $T_3 = O(\log n)$, and $d \log n = \mathcal{S}_{BDN}(m, n)$ it follows that $T_k = \mathcal{S}_{BDN}(m, n)$ for $k = 0, 1, 2, 3$.

The remainder of this section is devoted to showing that Steps (1) and (3) can be completed within the time claimed in the previous paragraph.

The following is a restatement of the main result of Section 5.5 using the language of this section.

Lemma 20 *Let $f : [0..a-1] \rightarrow 2^{[0..b-1]}$ be the function corresponding to a $(m, p, \lambda, d, c, \sigma)$-generalized expander G, let $S \subseteq [0..a-1]$ be of size at most n, and let q_G denote the quantity $(2/\lambda)(\sigma(d/2))^{-1}$. There exists an algorithm that constructs a $(\lfloor d/2 \rfloor + 1, q_G)$-set for S with respect to f running in time*

$$O(d \log n + \log n \log\log n(\log\log \Delta(G) - \log\log q_G))$$

on the BDN (V, E_{route}).

Lemma 21 *Let X be a subset of $A^{(k)}$ of size at most n_k. A $(\lfloor r/2 \rfloor + 1, q_1)$-set for X with respect to $\psi_k[A^{(k)}, B_l^{(k)}]$ can be constructed in $O(\mathcal{S}_{BDN}(m, n))$ time.*

Proof The proof follows from the preceding lemma and the fact that the graph $G(\psi_k[A^{(k)}, B_l^{(k)}])$ is isomorphic to $G_k^{(1)}$. □

A similar result holds for θ_k.

Lemma 22 *Let Y be a subset of $B_l^{(k)}$ of size at most n_k. An $(\lfloor d/2 \rfloor + 1, q_2)$-set for Y with respect to $\theta_k[B_l^{(k)}, C_l^{(k)}]$ can be constructed in $O(\mathcal{S}_{BDN}(m,n))$ time.*

So Step (1) will be carried out as follows. Partition X into N_k disjoint subset $X_0, \ldots, X_N$ of equal size. By Lemma 21 we can construct an $(\lfloor r/2 \rfloor + 1, q_1)$-set $\tilde{X}_i$ for each X_i with respect to $\psi_k[A^{(k)}, B_i^{(k)}]$ in $O(\mathcal{S}_{BDN}(m,n))$ time. The union $\tilde{X} = \cup_{i=1}^{N} \tilde{X}_i$ is a $(\lfloor r/2 \rfloor + 1, q_1)$-set for X with respect to ψ_k. All of these substeps may be completed in $O(\mathcal{S}_{BDN}(m,n))$ time.

Now the set Y produced by the algorithm during Step (2) has the property that it includes at most n_k integers from each $B_l^{(k)}$.

Step (3) will involve partitioning Y into subsets $Y_0, \ldots, Y_N$ where $Y_i = Y \cap B_i^{(k)}$. By Lemma 21 a $(\lfloor d/2 \rfloor + 1, q_2)$-set $\tilde{Y}_i$ for Y_i with respect to $\theta_k[B_i^{(k)}, C_i^{(k)}]$ can be constructed in $O(\mathcal{S}_{BDN}(m,n))$ time. The union $\cup_{i=1}^{N} \tilde{Y}_i$ is a $(\lfloor d/2 \rfloor + 1, q_2)$-set for Y with respect to θ_k. Again all of these substeps can be completed in $O(\mathcal{S}_{BDN}(m,n))$ time. This completes the description of the space efficient simulations for the BDN model and the proof of Theorem 1.

We now turn our attention to the question of adapting the techniques of this chapter to provide space efficient simulations on the AMOT model. The scheme is essentially the same as that presented above except for the choice of the graphs $G_k^{(2)}$ and the changes that result from this choice.

In the case of the AMOT the graph $G_k^{(2)} = (U, V, E)$ is a $(m_{k+1}, n_k, \lambda', d', d'/r', \sigma')$-generalized expander where λ' is a constant less that one, d' the degree of each vertex of U is at most $\beta \log(m/n)$ (β a constant independent of m and n), $r' = \log\log(m/n) - \log\log\log n$, and the threshold function σ' is defined as $\sigma'(k) = 1/d' 2^{d'/k}$.

The algorithm and its analysis mirrors that presented above except that Step (2) constructs a $(\lfloor r'/2 \rfloor + 1, q_1')$-set where $q_1' = (8/\lambda')r'$, and Step (5) constructs a $(\lfloor d'/2 \rfloor + 1, q_2')$-set where $q_5' = (8/\lambda')d'$. With this change the overall running time of the algorithm becomes

$$O\left(d' + \log n + q_1' q_2' + \log n \log\log n(\log\log(m/n) - log \log\log n)\right)$$

time which is

$$O\left((\log(m/n) + \log n \log\log n)(\log\log(m/n) - \log\log\log n)\right).$$

This completes the proof of Theorem 3.

Chapter 8

Lower Bounds

In this chapter we present a lower bound for the problem of simulating an (m, n)-PRAM on a p-node BDN. The next section presents an informal argument that captures several of the key ideas of the more formal lower bound given in the succeeding section. The formal argument is a refinement of the results of [AHMP87] and [KU88].

8.1 An Informal Bound

The simulations presented in this thesis satisfy the following four conditions.

C1 The number and locations of the copies of the individual variables do not vary with time;

C2 The simulation algorithm always reads or writes a majority of the copies of the variables it wishes to access;

C3 The simulations are *point-to-point*: in order to update a number of copies of a particular variable, a separate packet must be dispatched to update each; these packets are treated as distinct entities by the network, they can not be copied or combined except at the node that originally generated them; and

C4 The simulations are *on-line*: the simulation of each PRAM step is independent of future steps, in other words the simulator does not analyse the algorithm in order to "predict" future steps.

The simulations in the literature [UW87,AHMP87] can essentially be recast to satisfy these conditions without affecting their running times.

In this section we present an informal lower bound argument for simulations that obey all four conditions **C1-C4**. The more formal lower bound presented in the next section applies to all simulations obeying conditions **C3** and **C4**, *i.e.* to on-line, point-to-point simulations.

The core of the argument is the observation that if the simulation maintains only a small number of copies of each variable, then it will always be possible to find an n-tuple of variables that is impossible to read without causing huge memory congestion. On the other hand, the more copies per variable the more work involved in moving copies around the network during writing steps. Thus the lower bound hinges on the following trade-off, which has already appeared in the simulations of the previous chapters: memory congestion decreases with the number of copies per variable, whereas network traffic increases with the same quantity.

First we characterize the worst-case memory congestion in terms of the number of copies per variable using an argument that has appeared in various forms in the literature [NM82,UW87].

Assume that each variable has exactly r copies and that $m \geq p \geq n$. Consider the set of all subsets of V (the set of nodes in the BDN) of a given size $k \geq 2r$. There are $\binom{p}{k}$ such subsets in all. A variable is said to be *trapped* in a set $S \subseteq V$ if all of its copies lie in the nodes of S. Each variable is trapped in exactly $\binom{p-r}{k-r}$ of these subsets. Thus some subset of size k must trap at least

$$\frac{m\binom{p-r}{k-r}}{\binom{p}{r}} = \frac{m[k]_r}{[p]_r} \geq m\left(\frac{k-r+1}{p}\right)^r \geq m\left(\frac{k}{2p}\right)^r$$

variables since $k \geq 2r$. In particular some subset S of size k must trap n variables if

$$k \geq 2p\left(\frac{n}{m}\right)^{1/r} + r.$$

Thus accessing $r/2$ copies apiece for the variables in the set S will require at least

$$r/2 + \frac{n}{2p\left(\frac{n}{m}\right)^{1/r} + r.} \geq r/2 + (1/2)\min\left\{\left(\frac{n}{2p}\right)(m/n)^{1/r}, n/r\right\} \tag{8.1}$$

time.

Gottleib and Kruskal [GK84] have shown that any algorithm that routes any permutation of N items distributed evenly among the p nodes of a BDN must require at least

$$\Omega\left(\lceil (N/p)\rceil \log p\right) \tag{8.2}$$

time. In particular one would intuitively expect the writing of $r/2$ copies each for n variables to require $\Omega(\lceil r(n/p)\rceil \log p)$ time.

The combination of (8.1) and (8.2) suggests that any simulation that obeys conditions **C1-C4** above should have slowdown of at least some constant times the quantity

$$\min_{1\leq r\leq\infty}\left\{\min\left[(n/p)\left(\frac{m}{n}\right)^{1/r}, n/r\right] + \lceil r(n/p)\rceil \log p + r\right\} \tag{8.3}$$

which is at least some constant times

$$\min\left[\min_r\left\{(n/p)(m/n)^{1/r} + \lceil r(n/p)\rceil \log p + r\right\},\right.$$
$$\left.\min_r\left\{n/r + \lceil r(n/p)\rceil \log p + r\right\}\right].$$

Straightforward arithmetic manipulations show that the first term in the above expression is $\Omega(\log n \log(m/n)/\log\log p)$, and that the second term is bounded below by $\Omega(\sqrt{n}\left(\sqrt{(n/p)\log p}+1\right))$. Thus the above expression is

$$\Omega\left(\min\left[\frac{(n/p)\log p\log(m/n)}{\log\log p}, \sqrt{n}\left(\sqrt{(n/p)\log p}+1\right)\right]\right).$$

The formal lower bound embodying these ideas is more complex, but eventually distills down to a expression similar to (8.3) capturing a tradeoff on the number of copies per variable between the memory congestion and the routing overhead.

8.2 A Formal Lower Bound

A *straight-line* PRAM program consists of a sequence of PRAM steps without any looping or branching constructs. In [AHMP87] and [KU88] the following lower bound was established independently.

Theorem 8 *Let $m = \Omega(n^{2+\epsilon})$ and $T \geq (1+\epsilon)(m/n)$ for some fixed $\epsilon > 0$. For any on-line, point-to-point simulation of an (n,m)-PRAM on a BDN with n nodes, there is a straight-line SIMD PRAM program running for T steps on an (n,m)-PRAM whose simulation requires time*

$$\Omega\left(T min\left[\frac{\log m \log n}{\log\log m}, \sqrt{n\log n}\right]\right).$$

The above result makes no assumption about the number or placement of copies in the network. By refining the proof technique of [AHMP87,KU88], we extend their result to cover the range of $m \leq n^{2+\epsilon}$ and to apply where $p \geq n$.

Theorem 2 *Let $T \geq (1+\epsilon)(m/n)$ for some constant $\epsilon > 0$ and let $m \geq p \geq n$. For any on-line, point-to-point simulation of an (n,m)-PRAM on a BDN with p nodes, there is a straight-line SIMD PRAM program running for T steps* *on an (n,m)-PRAM whose simulation requires time*

$$\Omega\left(T(n/p)\min\left[\frac{\log^2(m/n)}{\log\log(m/n)},\sqrt{p\log(m/2p)}\right]\right).$$

We first review the ideas behind the proof of Theorem 8, and then indicate the modifications necessary to obtain Theorem 2. For ease of reference, we will use the same notation as in [AHMP87].

The approach consists of constructing an adversary straight-line program of $(1+\epsilon)(m/n)$ instructions, the first m/n of which are used to initialize all m PRAM variables, and the remaining $\epsilon m/n$ form $\tau \triangleq \epsilon m/2n$ *stages* each consisting of a read instruction followed by a write instruction. The variables to be accessed in each of these instructions are chosen, by the adversary, according to the state of the BDN memory so as to make each read and write as time-consuming as possible. The goal of the simulator is to simulate the PRAM steps generated by the adversary as efficiently as possible.

Both the on-line assumption and the point-to-point assumption play key roles in the lower bound. The point-to-point assumption implies that if a processor at node a wishes to update copies of some variable stored at nodes $b_1, b_2, \ldots, b_r$ then this update will require $\Omega(\sum_{i=1}^{r}\delta(a,b_i))$ processor cycles to complete where δ is the usual graph-theoretic distance function. The on-line assumption also plays a crucial role in the lower bound argument. Theorem 2 implies that there exists a sequence of PRAM instruction for which any on-line point-to-point simulation requires the slowdown stipulated in Theorem 2. It is conceivable that in simulating a PRAM *algorithm* rather than an arbitrary sequence of PRAM steps, an analysis of the algorithm could expose some structure that could be exploited to provide faster simulations.

In order to construct a sequence of hard reads and writes, [AHMP87,KU88] define the notion of the *redundancy* of a variable. Loosely speaking, the idea is as follows. During the simulation of a write step, each writing processor will dispatch copies to various nodes in the network. For the purpose of analysing the cost of a write step, only those which travel to nodes at least a certain minimum distance q from the writing processor are counted, the others are considered free. It turns out that when considering the cost of a read step the free copies can also be ignored, since the free copies of many variables will be concentrated in a small number of nodes, and consequently at most a small number of processors can read free copies without causing massive congestion. In order to formalize some of these notions, denote by Γ_u^t the set of nodes which contain a valid copy of the variable u at the start of the t^{th} stage. Let $G = (V, E)$ be the graph representing the structure of the network. For each node $v \in V$, let $B(v)$ represent the set $\{x : \delta(v, x) \leq q\}$ of nodes, where q is a parameter to be specified later. The *redundancy* of a variable u is defined as follows:

$$r_u^t \triangleq \min_{v \in V} |\Gamma_u^t - B(v)|.$$

The average redundancy of the entire set of variables is $r^t \triangleq \left(\sum_{u \in U} r_u^t\right)/m$.

First we will characterize the worst-case simulation time of a read step, where r is the average redundancy at the beginning of the step. Let $g(r)$ denote this quantity.

While [AHMP87,KU88] obtain a function g for all $m = \Omega(n^{2+\epsilon})$, we shall obtain a different g for all m. The two following refinements of the proof technique enable its application to the lower range of m: (i) the value of q is chosen as a function of both m and n, instead of just n, and (ii) in constructing the set of variables which are hard to read, the union of several balls is considered, rather than just one ball. The details follow.

Lemma 23 *Let d be the maximum degree of a node in the BDN. For $n \leq p \leq m$, if at the beginning of a read step the average redundancy is r, then there exists a set of n variables that will require*

$$g(r) = \Omega\left(\min\left[(n/p)(m/2p)^{1/(8r+2)}, n/\lfloor 2r \rfloor\right]\right)$$

time to be read by the simulating BDN.

Proof We may assume without loss of generality that (i) $r \leq (1/16)(\log(m/2p)$, (ii) $r \leq p/8$, and (iii) $m \geq 2d^9p$. To justify this notice that $r > (1/16)\log(m/2p)$ implies that $g(r) \leq (m/2p)^{16/\log(m/2p)} = 2^{16}$, that if $r > p/8$ then $n/r < 8(n/p) \leq 8$, and that $m < d2^9p$ implies that $m/2p$ is at most $d2^8$.

We will select the value of q to be $\lfloor \log(m/2p)/(2\log d) \rfloor - 1$. Note that since $m \geq 2d^4p$ we have $q \geq 1$. Also, for this particular choice of q,

$$|B(v)| \leq d^{q+1} \leq (m/2p)^{1/2},$$

for all $v \in V$. Let $U' = \{x \in U : r_x \leq 2r\}$. (Here the r_x represents the redundancy of x at this point in the simulation.) Clearly $|U'| \geq m/2$. Partition U' into disjoint sets X_v, $v \in V$, so that

$$X_v = \{x \in U' : r_x = |\Gamma_x - B(v)|\}.$$

(Note that some of the X_v may be empty.) We will deal separately with the case $r < 1/2$ and the case $r \geq 1/2$.

If $r < 1/2$, then all the elements of U' have redundancy 0, since each individual variable has redundancy strictly less than one and individual variables can only have integral redundancies. Consider a ball $B(v)$ such that $|X_v|$ is as large as possible, which implies that $|X_v| \geq \lceil m/2p \rceil$. Clearly reading the $\lceil m/2p \rceil$ variables in X_v will require at least

$$\frac{|X_v|}{|B(v)|} \geq \frac{\lceil m/2p \rceil}{(m/2p)^{1/2}} = \Omega((n/p)(m/2p)^{1/2})$$

steps. This establishes the proof in the case where $r < 1/2$.

Now suppose that $r \geq 1/2$. Let

$$l = n \Big/ (m/2p)^{3/4} ,$$

and let $L \subseteq V$ be the set of indices of the l largest X_v. Let $X_L = \bigcup_{v \in L} X_v$. Clearly

$$|X_L| \geq (m/2p)l.$$

We also let $B_L = \bigcup_{v \in L} B(v)$. Now

$$|B_L| \leq l(m/2p)^{1/2} \leq n \Big/ (m/2p)^{1/4} \leq n/4,$$

since $m \geq 2^9 p$.

Now for any $W \subseteq V$, let

$$A_W = \{x \in X_L : \Gamma_x \subseteq B_L \cup W\}.$$

Recall that each variable in U' has redundancy at most $2r$ and hence at most $\lfloor 2r \rfloor$ and so by the combinatorial argument given earlier for any k with $\lfloor 2r \rfloor \leq k \leq n$, there is a subset W of V with $|W| = k$ and

$$|A_W| \geq |X_L| \binom{p - \lfloor 2r \rfloor}{k - \lfloor 2r \rfloor} \Big/ \binom{p}{k} \geq |X_L| \left(\frac{k - \lfloor 2r \rfloor + 1}{p} \right)^{\lfloor 2r \rfloor}.$$

We now choose

$$k = \left\lceil p(n/|X_L|)^{1/\lfloor 2r \rfloor} + \lfloor 2r \rfloor - 1 \right\rceil.$$

Use of the chosen value of k in the lower bound for $|A_W|$ yields $|A_W| \geq n$. Then we can choose the set of variables to be read in the current step as any n-element subset of A_W.

Also

$$\begin{aligned} k &\leq p(n/|X_L|)^{1/\lfloor 2r \rfloor} + \lfloor 2r \rfloor \\ &\leq p \Big/ \left(\frac{m}{2p} \right)^{1/4\lfloor 2r \rfloor} + p/4. \end{aligned}$$

Here we have used the fact that $\lfloor 2r \rfloor \leq 2r \leq p/4$ by assumption (ii) above, and the relation $r \leq (1/16)\log(m/2p)$ which implies that $(m/2p)^{4\lfloor 2r \rfloor} \geq 4$. Thus $|B_L| + k \leq n/4 + p/2 < p$.

Since all of the valid copies of these variables are contained in $|B_L \cup W| \leq |B_L| + k$ nodes, it follows that the simulation of that read step (in the case where $r \geq 1/2$) will require at least

$$\begin{aligned}
\frac{n}{|B_L|+k} &\geq \frac{n}{n/(m/2p)^{1/4} + p/(m/2p)^{(1/4)/\lfloor 2r \rfloor} + \lfloor 2r \rfloor} \\
&\geq (1/3)\min\left[(m/2p)^{1/4}, (n/p)(m/2p)^{1/(8r+2)}, n/\lfloor 2r \rfloor\right] \\
&\geq (1/3)\min\left[(n/p)(m/2p)^{1/(8r+2)}, n/\lfloor 2r \rfloor\right]
\end{aligned}$$

time since $p \geq n$ and $r \geq 1$.

Combining the results for the two subcases $r < 1/2$, and $1/2 \leq r$ we obtain the stated result. □

Now let S represent the time required to simulate the T steps of the adversary program. This quantity may be bounded as follows.

Lemma 24 *With the above notation, the last τ stages of the adversary PRAM program can be chosen so that*

$$S = \Omega\left(T max((n/p)\hat{r}q, g(\hat{r}))\right),$$

where $\hat{r} = (1/\tau)\sum_{t=1}^{\tau} r^t$, is the average redundancy over the entire simulation.

Proof Intuitively, the term $\hat{r}q$ captures the cost of writing $\hat{r}$ copies outside a ball of radius q centred at the writing processor, and the term $g(\hat{r})$ is the cost of reading variables when there is an average of $\hat{r}$ copies per variable kept outside a given ball of radius q.

The adversary program is constructed to make each read and write operation as expensive as possible. Thus each read operation reads the n variables that will cause the worst memory congestion. Each write step is chosen to overwrite the n variables with the highest redundancy.

The t^{th} write step will decrease the total redundancy $\sum_{x \in U} r_x^t$ by at least nr^t. Thus the total redundancy is at most

$$\frac{Sp}{q} - \sum_{t=1}^{\tau} nr^t$$

since each counted copy originally cost q processor cycles to write and each write invalidates the old copies of that particular variable.

Since the total redundancy must be non-negative we can conclude that

$$S \geq \tau\hat{r}(n/p)q.$$

From the previous lemma we know that the read steps of the adversary program can be chosen so that

$$S \geq \sum_{t=1}^{\tau} g(r^t).$$

Thus

$$2S \geq \tau\hat{r}(n/p)q + \sum_{t=1}^{\tau} g(r^t).$$

We may now apply the following well known inequality: if f is a real function that is convex downwards, then for any set $\{x_1, x_2, \ldots, x_N\}$ of real numbers

$$(1/N)\sum_{i=1}^{N} f(x_i) \geq f\left((1/N)\sum_{i=1}^{N} x_i\right).$$

This inequality implies that

$$\sum_{t=1}^{\tau} g(r^t) \geq \tau g(\hat{r})$$

which when substituted into the above expression yields

$$\Omega\left(T[\hat{r}(n/p)q + g(\hat{r})]\right). \tag{8.4}$$

This completes the proof of the lemma. □

In conclusion,

$$\begin{aligned}
(p/n)ST &\geq \hat{r}q + \min\left[(m/2p)^{1/(8\hat{r}+2)}, p/\hat{r}\right] \\
&\geq \min\left[\min_r\left\{rq + (m/2p)^{1/(8r+2)}\right\}, \min_r\left\{rq + p/r\right\}\right] \\
&\geq \Omega\left(\min\left[\log^2(m/2p)/\log\log(m/2p), \sqrt{p\log(m/2p)}\right]\right).
\end{aligned}$$

Thus S, the minimum worst-case time to simulate T PRAM steps is at least

$$\Omega\left(T(n/p)\min\left[\frac{\log^2(m/2p)}{\log\log(m/2p)}, \sqrt{p\log(m/2p)}\right]\right)$$

which completes the proof of Theorem 2

This lower bound relies heavily on the fact that the simulator is assumed to be both point-to-point and on-line. It is conceivable that faster simulations are possible by violating one or other of these constraints.

For example, it principle at least, a non point-to-point simulation might reduce network traffic. One possible approach would be to cluster the r copies of each variable u in a small physical neighbourhood N_u of the network; to update u a processor would dispatch *one* message to some processor in N_u which would then generate r separate messages to update the individual copies of u which are nearby. The total amount of work involved in updating the copies could potentially be much less than $O(\log p)$ per copy. Whether this idea can be exploited to provide fast simulations remains to be seen. (See [AHMP87] for a simple simulation that is not point-to-point that has a simulation time of $O(\sqrt{n})$; this is competitive only when m is huge in relation to p and n.)

Chapter 9

Conclusion

At the heart of every parallel computation is some mechanism that allows processors to share information such as intermediate results and control information among themselves in order to collectively complete the assigned task. This pool of shared information is the glue that binds the actions of the various individual processors into a coherent computation. For the computation to be efficient it is critical that the processors be able to access the operands they need from this pool as quickly as possible. In this thesis we have considered the problem making a large collection U of shared variables accessible to processors in a bounded degree network. We have shown that it is possible to represent such a collection among n nodes of an parallel machine using a total amount of storage greater than $|U|$ by at most a factor polylogarithmic in $|U|/n$, in all such that any n-tuple of variables from U may be updated or probed efficiently. These simulations are the first deterministic simulations to be economical with respect to space; previous simulations used at least $\Omega(|U|n)$ storage in all. The algorithm is expressed in terms of a handful of general-purpose communication algorithms, principally sorting and the generalized routing algorithm presented in Chapter 2. The efficiency of the simulation algorithm depends on how efficiently the underlying network architecture supports these communication primitives. We have presented efficient algorithms for these primitives on two different networks: an expander-based bounded degree network and an augmented mesh-of-trees. The simulations presented here are faster than any previously known for these architectures in most cases. It has been shown that an arbitrary step of an (n, m)-PRAM may be simulated a n-node expander-based BDN in

$$\begin{aligned} \mathcal{S}_{BDN}(m,n) &= O(\log n \log(m/n)/\log\log n \\ &\quad + \log n \log\log n(\log\log(m/n) - \log\log\log n)) \end{aligned}$$

time in the worst-case. When $m = n(\log n)^{O(1)}$ (as is the case for many efficient parallel computations), then the running time of $O(\log n \log\log n)$ is within a small factor ($O(\log\log n)$) of the lower bound imposed by the diameter of the network and of the expected running time of the best randomized simulations that have appeared in the literature. It has also been shown that faster simulations are possible on more powerful machines. For example, an $n \times n$ AMOT can simulate an arbitrary step of an (n, m)-PRAM in

$$O\left((\log(m/n) + \log n \log\log n)(\log\log(m/n) - \log\log\log n)\right)$$

time.

Recent advances in the problem of sorting on such practical architectures hypercube [AH88,CS88,Pla89] suggest that efficient simulations may be possible on other widely-used machine architectures. Efficient algorithms for generalized routing for such architectures would translate into efficient PRAM simulation algorithms.

Overall these results suggest that, in principle at least, it is feasible to provide the abstraction of a shared memory on distributed models of parallel computation with reasonable worst-case performance. However, the complexity of the algorithm, the nonconstructive nature of the memory organizations used, and the large constants in the expressions for the running times of the algorithms mean that these results are unlikely to practical in their present form. These results provide processors in a parallel machine a tool with which processors to control congestion. Some development of these ideas might prove useful in a context where the penalty for congestion is very large in relation to the speed of the processors, such as in the case of a network of fast processors attached to an array of slow discs.

The most important open problem is that of providing an efficient explicit construction for the memory organizations of the type employed here. As we have mentioned earlier this is related to a open graph-theoretic question of the explicit construction of concentrators. Such a construction would greatly simplify the problem of achieving space efficient simulations, and would make the simulations uniform.

The results of this thesis limit the possible gap between running time T_M^Π of the best algorithm for a problem Π on a machine M and the running time $\mathcal{S} \cdot T_{PRAM}^{\Pi}$ of a simulation of the best PRAM algorithm for Π. Many problems of interest however exhibit some structure that may be exploited to obtain efficient algorithms for a particular architecture. An excellent example is provided by the FFT algorithm on the butterfly network. Since simulations of PRAM program are essentially blind to any structure in the problem, the ratio

$$\Lambda_\Pi \triangleq \frac{T_M^\Pi}{T_{PRAM}^\Pi}$$

measures the inefficiency of the simulation. The question of whether

$$\Lambda_\Pi = \Omega(\mathcal{S}_{BDN}(m, n))$$

for any "natural" problems Π remains open.

Bibliography

[AH88] A. Aggarwal and M.A. Huang. Network complexity of sorting and graph problems and simulating CRCW PRAMs by interconnection networks. In *Proceedings of the 3^{rd} Aegean Workshop on Computing, Corfu, Greece*, pages 339–350, June 1988.

[AHMP87] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal of Computing*, 16(5):808–835, Oct 1987.

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

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15^{th} Annual Symposium on the Theory of Computing*, Boston, Massachusetts, pages 1–9, Apr 1983.

[BH85] A. Borodin and J.E. Hopcroft. Routing, merging, and sorting on a parallel models of computation. *Journal of Computer and System Sciences*, 30(1):130–144, Feb 1985.

[BS78] G. Baudet and D. Stevenson. Optimal sorting algorithms for parallel computers. *IEEE Transactions on Computers*, c-27(1):84–87, Jan 1978.

[CS88] R. Cypher and J.L.C. Sanz. Cubesort: An optimal sorting algorithm for feasible parallel computers. In *Proceedings of the 3^{rd} Aegean Workshop on Computing, Corfu, Greece*, pages 456–464, June 1988.

[Csa76] C. Csanky. Fast parallel matrix inversion algorithms. *SIAM Journal of Computing*, 5(4):618–623, Dec 1976.

[CT65] J.M. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, Apr 1965.

[DDPW83] D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson. Superconcentrators, generalizers, and generalized connectors with limited depth. In *Proceedings of the* 21^{st} *Annual ACM Symposium on the Theory of Computers, Seattle, Washington*, pages 42–51, May 1983.

[DKM+88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Hiede, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings of the* 29^{th} *Annual Symposium on the Foundations of Computer Science, White Plains, New York*, pages 524–531, Oct 1988.

[FKS84] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, Jul 1984.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the* 10^{th} *Annual Symposium on the Theory of Computing*, San Diego, California, pages 114–118, May 1978.

[GG81] O. Gabber and Z. Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22:407–420, 1981.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the* 7^{th} *ACM Symposium on Operating Systems Principles, Pacific Grove, California*, pages 150–159, Dec 1979.

[GK84] A. Gottlieb and C.P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *Journal of the ACM*, 31(2):193–209, Apr 1984.

[Gol78] L. Goldschlager. A unified approach to models of synchronous parallel machines. In *Proceedings of the* 10^{th} *Annual Symposium on the Theory of Computing*, San Diego, California, pages 89–94, May 1978. Also appears in *Journal of the ACM* 29(3):1073-1086, July 1982.

[Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, Massachusetts, 1969.

[HB88] K. T. Herley and G. Bilardi. Deterministic simulations of PRAMs on bounded-degree networks. In *Proceedings of the* 26^{th} *Annual Allerton Conference on Communication, Control and Computation, Monticello, Illinois*, pages 1084–1093, Sept 1988. Also available as technical report TR 88-951, Computer Science Department, Cornell University, November 1988.

[Her89a] K. T. Herley. Efficient simulations of small shared memories on bounded degree networks. In *Proceedings of the* 30^{th} *Annual Symposium on the Foundations of Computer Science, Research Triangle Park, North Carolina*, pages 390–395, Oct 1989.

[Her89b] K. T. Herley. Improved bounds for the token distribution problem. Technical Report TR 89-1051, Computer Science Department, Cornell University, October 1989.

[Hir76] D. S. Hirshberg. Parallel algorithms for the transitive closure and connected components problems. In *Proceedings of the* 18^{th} *Annual Symposium on the Theory of Computing, Herschey, Pennsylvania*, pages 55–57, May 1976.

[Hir78] D.S. Hirschberg. Fast parallel sorting algorithms. *Communications of the ACM*, 21(8):657–661, Aug 1978.

[HMP89] T. Hagerup, K. Mehlhorn, and B. Parker, Jan 1989. Private Communication.

[HP89] S. W. Hornick and F. P. Preparata. Deterministic PRAM simulations with constant redundancy. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Sante Fe, New Mexico*, June 1989.

[KR88] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. Technical Report UCB/CSD 88/408, Computer Science Division, University of California, Berkeley, California, Mar 1988.

[KRS85] C.P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, c-34(10):965–968, Oct 1985.

[KU88] A. Karlin and E. Upfal. Parallel hashing: an efficient implementation of shared memory. *Journal of the ACM*, 35(4):876–892, Oct 1988.

[Lei81] F. T. Leighton. New lower bound techniques for VLSI. In *Proceedings of the 22^{nd} Symposium on the Foundations of Computer Science,* Nashville, Tennessee, pages 1–12, Oct 1981.

[Lei85] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, c-34(4):344–354, April 1985.

[LM89] T. Leighton and B. Maggs. Expanders might be practical: Fast algorithms for routing around faults in multibutterflies. In *Proceedings of the 30^{th} Annual Symposium on the Foundations of Computing, Research Triangle Park, North Carolina*, pages 384–389, Oct 1989.

[LPP87] F. Luccio, A. Pietracaprina, and G. Pucci. A new scheme for deterministic simulations of PRAMs in VLSI. Technical Report S-87-11, Dipartmento di Informatica, Università di Pisa, June 1987.

[LPP88] F. Luccio, A. Pietracaprina, and G. Pucci. A probabilistic simulation of PRAMs on a bounded degree network. *Information Processing Letters*, 28(3):141–147, July 1988.

[LPS86] A. Lubotzky, R. Phillips, and P. Sarnak. Explicit expanders and the Ramanujan conjectures. In *Proceedings of the 18^{th} Annual Symposium on the Theory of Computing,* Berkeley, California, pages 240–246, May 1986.

[LPV81] G.F. Lev, N. Pippenger. and L.G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, c-30(2):93–100, Feb 1981.

[LV82] D. H Lawrie and C. R. Vora. The prime memory system for array access. *IEEE Transactions on Computers*, c-31(5):435–442, May 1982.

[Mey86] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal of Computing*, 15(1):106–119, Feb 1986.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21(4):339–374, Nov 1984.

[NM82] S. Nakamura and G.M. Masson. Lower bounds on crosspoints in concentrators. *IEEE Transactions on Computers*, c-31(12):1173–1179, Dec 1982.

[NMB83] D. Nath, S. N. Maheshwari, and P.C.P. Bhatt. Efficient VLSI networks for parallel processing based on orthogonal trees. *IEEE Transactions on Computers*, c-32(6):569–581, June 1983.

[Pla89] G. C. Plaxton. Load balancing, selection, and sorting on the hypercube. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, Sante Fe, New Mexico*, June 1989.

[Pre78] F. P. Preparata. New parallel-sorting schemes. *IEEE Transactions on Computing*, c-27(7):669–673, Jul 1978.

[PU89] D. Peleg and E. Upfal. The token distribution problem. *SIAM Journal on Computing*, 18(2):229–243, April 1989.

[PV81] F.P. Preparata and J. Vuilleman. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, 24(5):300–309, May 1981.

[PVW88] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proceedings of the* 10^{th} *International Conference on Automata, Languages, and Programming*, pages 597–609, 1988. Lecture Notes in Computer Science, Vol. 154, Springer Verlag.

[Ran87] A.G. Ranade. How to emulate shared memory. In *Proceedings of the* 28^{th} *Annual Symposium on the Foundations of Computer Science,* Los Angeles, California, pages 185–194, Oct 1987.

[Sch66] J. Schwartz. Large parallel computers. *Journal of the ACM*, 13(1):25–32, Jan 1966.

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, Oct 1980.

[Sie89] A. Siegel. On universal classes of fast high-performance hash functions, their time-space trade-off, and their applications. In *Proceedings of the* 30^{th} *Annual Symposium on the Foundations of Computing, Research Triangle Park, North Carolina*, pages 20–25, Oct 1989.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, c-20(2):153–161, Feb 1971.

[SV81] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2(1):88–102, Mar 1981.

[Tan84] R. M. Tanner. Explicit concentrators from generalized N-gons. *SIAM Journal of Algebraic and Discrete Mathematics*, 5(3):287–293, Sept 1984.

[Tho79] R.H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, Jun 1979.

[Upf84] E. Upfal. A probabilistic relation between desirable and feasible models of parallel computation. In *Proceedings of the* 16^{th} *Annual ACM Symposium on the Theory of Computing*, Washington, D.C., pages 258–265, May 1984.

[Upf89] E. Upfal. An $O(\log n)$ deterministic packet routing scheme. In *Proceedings of the* 21^{st} *Annual ACM Symposium on the Theory of Computers, Seattle, Washington*, pages 241–250, May 1989.

[UW87] E. Upfal and A. Wigderson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, Jan 1987.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4(1):45–50, 1983.

[VW83] U. Vishkin and A. Wigderson. Dynamic parallel memories. *Information and Control*, 56:174–182, 1983.