OPTICAL COMMUNICATION FOR POINTER BASED ALGORITHMS

by

Richard J. Anderson[1]
Gary L. Miller[2]

Technical Report CRI 88-14

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0782

Abstract

In this paper we study the *Local Memory P-RAM*. This model allows unit cost communication but assumes that the shared memory is divided into modules. This model is motivated by a consideration of potential optical computers. We show that fundamental problems such as list-ranking and parallel tree contraction can be implemented on this model in $O(\log n)$ time using $n/\log n$ processors. To solve the list-ranking problem we introduce a general asynchronous technique which has relevance to a number of problems.

[1]Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195, USA. Supported by a Presidential Young Investigator Award.

[2]Department of Computer Science, University of Southern California, Los Angeles, California 90089-0782, USA. Supported by NSF Grant DCR-8514961.

Optical Communication for Pointer Based Algorithms

Richard J. Anderson[*] Gary L. Miller[†]

(Extended Abstract)

Abstract

In this paper we study the *Local Memory P-RAM*. This model allows unit cost communication but assumes that the shared memory is divided into modules. This model is motivated by a consideration of potential optical computers. We show that fundamental problems such as list-ranking and parallel tree contraction can be implemented on this model in $O(\log n)$ time using $n/\log n$ processors. To solve the list-ranking problem we introduce a general asynchronous technique which has relevance to a number of problems.

1 Introduction

We consider a model of parallel computation that is especially suited to pointer based computation. We motivate this model by showing that basic problems, like list-ranking and parallel tree contraction, can be performed in $O(\log n)$ time using only $n/\log n$ processors. We also show that any step on this model can be simulated in unit time on this model by a machine with an optical communication architecture. Thus we contend that the basic problem of list-ranking can be performed on a feasible machine in $O(\log n)$ time, using $n/\log n$ processors. At the present time fixed connection machines such as the Connection Machine require $O(\log^2 n)$ time. We call our abstract machine the *Local Memory PRAM.*

The two most important models for polylogarithmic parallel algorithms have been the P-RAM and the fixed connection architecture such as the hypercube and the butterfly. The P-RAM model is usually thought of an abstract machine while the fixed connection machine is viewed as buildable or feasible.

Devising parallel algorithms for the P-RAM model is very convenient. One does not have to get involved in how read/write requests are actually performed on the P-RAM. Thus one

[*]Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195, USA. Supported by a Presidential Young Investigator Award.

[†]Department of Computer Science, University of Southern California, Los Angeles, California 90089, USA. Supported by NSF grant DCR-8514961

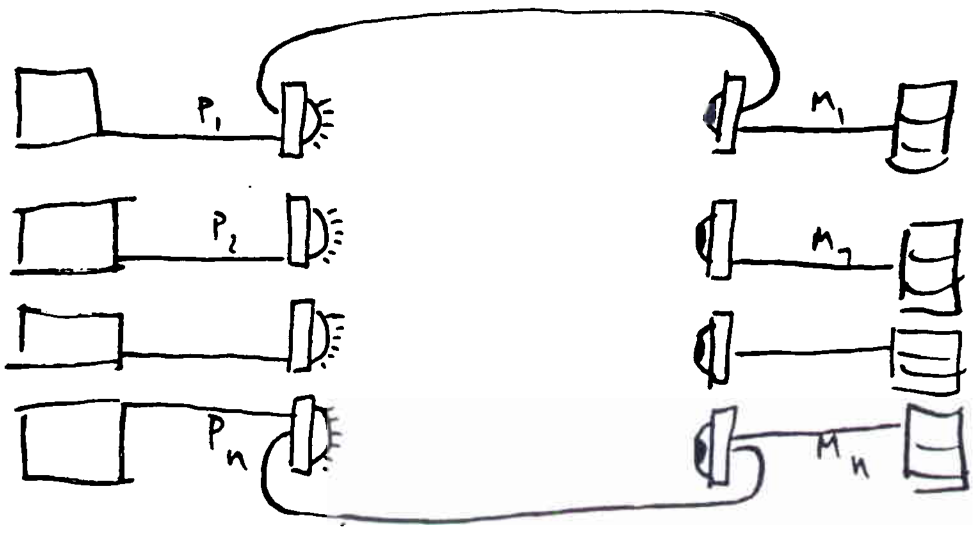

Figure 1: A Machine Based on Optically Communicating Processors

is able to concentrate on the intrinsic parallelism in the problem. There is a cost for ignoring how messages are finally routed between processors: For all known simulations it takes a factor of $\log n$ steps per step of the P-RAM to simulate the step on a fixed connection architecture. Many researcher contend that the only feasible parallel machine is a fixed connection architecture of bounded degree [UW87,KU86,Ran87]. We argue that feasible models of computation can be constructed that are of unbounded degree based on optical devices.

We present a possible physical parallel computer that uses unbounded fanout. We give a diagram of the device in Figure 1. The machine consists of a collection of processors and a collection of memory modules. Each processor and each memory module has one optical receptor and one optical transmitter. Each transmitter can be focused on a receptor in unit time. We assume that no two beams interfere unless they are focused on the same receptor. We shall assume that each processor is itself a RAM. Each memory module also has a direct connection back to one of the processors, so that it can be thought of both as global memory and as the memory for one of the processors.

The main assumptions we make about our model are:

1. Photons or light beams do not interact.

2. Light travel between two points in unit time.

3. Photons can be redirected in unit time.

The first assumption if not correct is at least close enough given the quality of the other assumptions. The second assumption is similarly implicit in the RAM model and the VLSI work of Thompson [Tho80]. The third assumption is the most critical one since the time to redirect the beam may be very large compared to the other constants involved in an actual device.

Two methods used today to redirect light are both quite slow. The first method consists of a rotating mirror. The second method involves acoustically vibrating a crystal which is placed in the path of the light beam. The devices are called acousto-optic cells, see [SJRV87,Bel86]. For both methods the refocusing times are in microseconds.

However, we see no intrinsic reason why the redirection time can not be in nanoseconds for a few thousand different directions. Reif has suggested a method for redirection of light based on holography and coherent systems, which will redirect beams at the speed of light rates [Rei]. Reif and Johnson at the Center for Optical-Electron Systems are preparing experiments on these new ideas. If such a device, as proposed by Reif, can be built, our optical computer will be a real possibility. The purpose of these comments is to indicate that it may be possible to build devices with the property that arbitrarily large number of processors may communicate in unit time delay.

The difficulty in proposing a model as feasible is that it seems to require from us an in depth study of the present knowledge of both physics and engineering. We do not have the knowledge in either field to make such a judgement. We do not believe this is always necessary. Accepted models such as the *RAM* are not feasible in the sense that they cannot be simulated in real time by a device that satisfies all the known physical laws [CR73]. A models importance lies in the fact that many important algorithms may be easily understood in the model. Thus we contend that first, our model is as feasible as the *RAM* and the fixed connection architecture, and second, our model allows us to introduce important new algorithmic ideas.

We define a formal model for a machine that consists of a collection of optically communicating RAMs below:

Definition 1.1 *A Local Memory P-RAM is a collection of RAMs which share a common message passing architecture. This architecture satisfies the following rules:*

- *At any time a processor can send at most one message. Its destination is another processor.*
- *The message will succeed in reaching the processor if it is the only message with that processor as it destination at that time step.*
- *All messages succeed or fail (and thus are discarded) in unit time.*

To insure that every processor knows when its message succeeds we assume that the Local Memory P-RAM is run in two phases. In the first phase, read/write messages are sent, and in the second, values are returned to successful readers and acknowledgements are returned to successful writers. Note that since each processor successfully sends and receives at most one message from the first phase, all the messages of the second phase succeed.

It should be clear that a Local Memory P-RAM with m processors can simulate, in real time, an EREW P-RAM using P processors and n memory locations for $m = \max\{P, n\}$. On the other hand, a P processor EREW P-RAM can simulate in real time any P processor Local Memory P-RAM. Thus the important and new cases are when the memory is large.

A model similar to our Local Memory P-RAM was introduced by Mehlhorn and Vishkin [MV84] and further studied by Upfal and Wigderson [UW87]. The main difference between our work and theirs is a difference in motivation. They were primarily interested in the simulation of P-RAMs by fixed connection machines, and introduced a local memory model as an intermediate step in the simulation. They consider the problem of how to simulate a P-RAM on a local memory machine. Our primary interest is however, designing algorithms directly for a Local Memory P-RAM. We are interested in algorithms that are superior to the algorithms that arise directly from simulation.

2 Results

In this paper we study the power of the Local Memory P-RAM when the problem size is much larger than the number of processors. For larger inputs we assume that the input is distributed evenly amongst the local memories of the processors. We show that for some important problems, such as list-ranking and expression evaluation, it is possible to construct optimal randomized algorithms. These results match the best P-RAM algorithms and show that having local memory does not incur a substantial penalty.

A contribution of this paper is to introduce a general framework for a type of parallel algorithm. The framework is to view the problem as a communication problem - i.e. a certain set of messages must get through in order for the computation to succeed. When ever a message gets through, a relatively simple local computation is done. We identify two particular communication schemes which cover a number of different problems. The division of a problem into a communication and a computation problem is significant. If certain communication schemes are robust enough to be widely used, then it will be possible to just plug routines in to the schemes to get algorithms. One interesting aspect of the schemes that we introduce is that they are asynchonous. It will be important to develop algorithms for machines that are not as fully synchronized as the P-RAM and this is a step in that direction.

3 Time Sharing

It is very desirable for parallel machines to efficiently simulate a larger machine of the same type. We refer to this as time sharing or time slicing. If we have a machine of size N then we would like to be able to simulate a single step of a machine of size KN in $O(K)$ time. Most of the P-RAM models have the property that this can be done by straightforward algorithms.

The time sharing problem is to simulate one step of the larger machine on the smaller machine. For the local memory model, we consider the natural mapping of processors and memories:

$$p'_i \to p_{\lfloor i/K \rfloor} \qquad m'_i \to m_{\lfloor i/K \rfloor}.$$

For a step of the large machine, we can construct a bipartite graph between $p_1, \ldots, p_N$ and $m_1, \ldots, m_N$. For each successful memory access by a processor p'_i to m'_j, we put in an edge $p_{\lfloor i/K \rfloor}$ to $m_{\lfloor j/K \rfloor}$. This gives us a bipartite graph with maximum degree K which can be edge colored with K colors. The coloring can be used to assign time slots to the memory requests to avoid contention. This however, is not a real solution, since it is not possible to compute the coloring quickly. The best general solution that we know of takes K^2 steps to simulate a single step of the larger machine.

Theorem 3.1 *One step of a local memory machine of KN can be simulated by K^2 steps on a machine of size N.* ∎

It is natural to ask whether or not this is the best possible simulation algorithm. We can show that it is best possible for a natural class of simulation algorithms. Suppose that we have fixed the mapping of processors and memories. The issue to deal with is how much computation are we allowed in order to decide which messages to send out. We can show an $\Omega(K^2)$ lowerbound if a processor makes it decision on when to send out its messages based only on the addresses of its messages. We refer to this as a local algorithm.

Theorem 3.2 *The simulation of one step of a local memory machine of size KN by a machine of size N requires K^2 steps when a local algorithm is used.* ∎

4 Algorithms

In this section we present algorithms for the local memory model. Our algorithms are loosely synchronized message based algorithms. In the problems that we examine, each processor has a number of tasks that must be completed, although the order of tasks is not specified. Each task requires accessing one or two memory locations. We view this process as taking place on a graph with edges between processors and memory modules. There is an edge from a processor to a memory module if the processor must communicate with the module. When a processor succeeds in communicating with the module, the appropriate computation can be made and the communication graph can be updated.

The most difficult part of the problem is to design an algorithm to allow the appropriate messages to get through. Once we have the message sending algorithm, we can just plug in relatively simple routines to do the computation. There are two separate message manipulation algorithms that we need. The first algorithm is for the case that a computation can be done when a message gets through from a processor to a memory. If a message is sent from p_a to m_b, then the edge $p_a \rightarrow m_b$ can be deleted from the graph. The second algorithm is for the case where sending messages from p_a to m_b and from p_b to m_c allows the edges $p_a \rightarrow m_b$ and $p_b \rightarrow m_c$ to be replaced by the edge $p_a \rightarrow m_c$. In both of the cases the goal is to remove all of the edges from the graph.

We describe our algorithms for the case where each processor has $\log n$ messages to send, and each memory module will receive $\log n$ messages. The algorithms work just as well when the number of messages to send is greater. We concentrate on $\log n$ message per processor, since this corresponds to the most interesting case for studying optimal algorithms for such problems as list-ranking. This makes it easy to compare this work with other work done on optimal parallel algorithms.

4.1 Edge Removal Algorithm

The following algorithm sends messages from processors to memories. A computation can take place when a message gets through. The algorithm operates in two stages. The first stage sends the majority of the messages using a randomness to control when messages are sent. Not all messages will get through on the first stage, so a cleanup stage is used to take care of the remainder of the edges.

Message Stage The main algorithm relies on sending messages randomly. In a phase, each processor randomly chooses a message and then sends the message with a certain probability. The goal is to do this in a manner that insures a reasonable number of messages get through at each phase. To achieve our $O(\log n)$ time goal we need to get $cn/\log n$ messages through each phase, for some constant c.

To motivate our algorithm, consider the case where each processor has exactly K messages left to send and each memory has exactly K messages still to receive. Suppose each processor sends a message at random. A message gets through to a memory if it is the only message sent to that memory in the phase. It is easy to show that the probability that a given memory gets a message is at least $(1 - \frac{1}{K})^{K-1} > \frac{1}{e}$. Unfortunately, there is a tendency for the number of messages at processors and memories to become unbalanced, so that the analysis breaks down. It is tempting to run the simple randomized algorithm even in the more general unbalanced case. This however fails because certain memories can become saturated with requests. If a memory that still must receive many requests is adjacent to processors with few messages left, the memory will receive several requests per phase, and not be able to answer any of them.

The basic problem with the randomized algorithm arises when a processor gets to many messages through. Our solution is to control the rates that messages are sent out. This is done by viewing the situation as being roughly balanced. We will refer to the set of unsent messages of a processor, and the set of unreceived message of a memory as a *queue*. In the k-th round of our algorithm, we shall act as if all queues have size between 2^k and 2^{k-1}. Messages are sent for $a2^k$ phases in the k-th round. We show that each queue has a reasonably high probability of being reduce to fewer than 2^{k-1} items by the round. A phase consists of each processor choosing a message and randomly sending it out. We do this in a manner so that if a message is in a queue of size at most 2^k it has probability $\frac{1}{2^k}$ of being sent out. If a processor does not succeed in sending out enough messages it quits sending messages and waits for the cleanup phase.

The code for the k-th round of the algorithm is given. The messages to be sent are stored in an array A. The variable *MaxMessage* gives the maximum message in the queue. The

```
$Round_k$
  for $i := 1$ to $a2^k$ do
    $j := RandomInteger(1, 2^k)$;
    if $j \leq MaxMessage$ then
      send$A[j]$;
      $A[j] := A[MaxMessage]$;
      $MaxMessage := MaxMessage - 1$;
  if $MaxMessage > 2^{k-1}$ then halt;
```

unsent messages occupy the first *MaxMessage* locations of A.

The full algorithm is to have rounds for k running from $\log\log n$ down to 1. We will now show that the expected number of messages left after the algorithm is done is bounded by $cn/\log n$. The following lemmas show that there is a good chance that a queue will be reduced during a round. A queue is said to be oversized if it contains more than 2^k items at the start of round k. Since a processor stops attempting to send messages when it becomes oversized, we can view the memory queues as never being adjacent to oversized processor queues. A processor can be adjacent to an oversized memory queue. When this occurs, the processors ability to successfully send messages degrades. However, we can get by by making a fairly pessimistic assumption about what happens to a processor queue when it is adjacent to an oversized memory queue. This allows us only to have to worry about the case where a processor is not adjacent to oversized memory queues. The following lemmas show that things work well when queues are balanced.

Lemma 4.1 *Suppose M is a memory with at most 2^k items in its queue. The probability that M becomes oversized in round k is at most $e^{-\beta_1 2^k}$ for some constant β_1 independent of k.* ∎

Lemma 4.2 *Suppose P is a processor with at most 2^k items in its queue and P is not adjacent to any oversized memories. The probability that P becomes oversized in round k is at most $e^{-\beta_2 2^k}$ for some constant β_2 independent of k.* ∎

Theorem 4.3 *The expected number of elements remaining after the algorithm is done is at most $cn/\log n$ for some constant c.*

Proof: If some message is not sent, then the message is in a processor's queue that became oversize at some point in time. We distinguish between two cases on how the queue became oversized - depending on whether or not it was adjacent to an oversized memory queue. If the processor's queue was not adjacent to an oversized memory queue, we bill the message to the round that the queue became oversized. If the processor queue was adjacent to an oversized memory queue, then we bill the item to the round which the processor first became adjacent to an oversized memory queue. The expected amount billed to round k is at most

$$n/\log n 2^k e^{-\beta_2 2^k} + n/\log n 2^{2k} e^{-\beta_1 2^k}.$$

The first term arises from processor queues not adjacent to oversize memory queues and the second arises form processor queues adjacent to oversize memory queues. The expected number of messages left is therefore at most:

$$n/\log n \sum_{1\leq k\leq \log\log n} \left(2^k e^{-\beta_2 2^k} + 2^{2k} e^{-\beta_1 2^k}\right) = cn/\log n.$$

∎

Cleanup Stage At the start of the cleanup stage, there may be as many as $cn/\log n$ messages that have not been resolved. The first step is to distribute the message sources equally amongst the processors. This is done by first computing a parallel prefix sum and then redistributing the source messages. Once the sources have been redistributed, the second step is to send all of the messages. Both the redistribution and the message sending take $c\log n$ phases. To avoid message collisions, in a phase, all messages with a given pair of local source and destinations are sent. In the redistribution step, the local sources range from 1 to $\log n$, while the local destinations range from 1 to c, and in the message step, the sources range from 1 to c and destinations range from 1 to $\log n$.

Applications The message routing algorithm removes all edges from the graph in $O(\log n)$ expected time. If we have an algorithm that uses this reduction algorithm and takes $O(1)$ processing time per edge removal, then it is also an $O(\log n)$ algorithm. We give two applications of this algorithm, reversing the links in a list and machine simulation in the local memory model.

When working with linked lists it is very often taken for granted that they are doubly linked. This is usually justified, since in models such as sequential or parallel RAMs, reversing links is trivial. The natural parallel algorithm for reversing the links is to have each processor reverse the links of the cells associated with the processor. However, this algorithm runs into contention problems when implemented on a local memory P-RAM. This example motivated our consideration of the general edge removal process, since each processor must get message through for each edge. When processor P_a succeeds in getting a message through to M_b, the message tells one of the cells in M_b to point to a cell in M_a. The pointer can be set by P_b.

The second example is for the simulation of one local memory machine by a smaller local memory machine. Our simulation is for simulating a machine $\mathcal{M}_1$ of size N by a machine $\mathcal{M}_2$ of size N/K where $K \geq \log n$. The mapping of processors and memories that the memory and processor i in $\mathcal{M}_1$ is simulated by memory and processor $\lfloor i/K \rfloor$ in $\mathcal{M}_2$. In the simulation, each processor of $\mathcal{M}_2$ simulates the reads and writes of several processors. This can be done by sending messages to the processors holding the correct memory modules. It is necessary to simulate the collisions that occur in a round $\mathcal{M}_1$. This is not that difficult; just a detail to watch out for.

4.2 Vertex Removal Algorithm

The second scheme that we consider reduces the graph by splicing out vertices. If P_a sends a message to M_b and P_b sends a message to M_c, then the edges $P_a \rightarrow M_b$ and $P_b \rightarrow M_c$ are removed and an edge $P_a \rightarrow M_c$ is added. A message from P_a to M_b has a next message from P_b to some memory. If the message gets through, then the next message is triggered and sent off. For technical reasons, we do not want sequences of messages removed in the same phase. We take care of this special case within the algorithm.

The algorithm is similar to the previous algorithm, except that two messages must get through successfully for a computation to be made. The innermost step is to have some processors send messages at random. The messages that make it through trigger the sending of a second set of messages. The key to the randomization is again to choose the appropriate rate of sending out messages. The algorithm is run until the number of elements is reduced to $cn/\log n$ and then a cleanup phase is run.

The analysis is similar to the previous theorem, although the details are more complicated. Some care is necessary in dealing with conditional probabilities arising form the two message phases in the proof of the lemma. The theorem is proved by considering the expected number of oversized queues. The analysis takes into account the interaction of oversized queues between rounds.

Lemma 4.4 *Suppose Q is a queue (processor or memory) with at most 2^k items at the start of round k. If Q is not adjacent to any oversized queues, the probablility that that Q becomes oversized in round k is at most $e^{-\beta 2^k}$ for some constant β independent of k.* ∎

Theorem 4.5 *The expected number elements remaining after the algorithm is done is at most $cn/\log n$ for some constant c.* ∎

Applications The main application of this reduction algorithm is for an optimal randomized list-ranking algorithm on the local memory model. The list ranking problem is: given a linked list in memory, compute the distance that each cell is from the end of the list. The problem is a fundamental data structure problem, and has many applications. The list ranking problem has been studied extensively with the goal of getting $n/\log n$ processor, $O(\log n)$ time P-RAM algorithms [Wyl79,CV86]. We show that these bounds can be met on the local memory model for a randomized algorithm.

If we have n processors available, then it is easy to solve the list-ranking problem by a path doubling approach in $O(\log n)$ time. The idea that has been used to get optimal algorithms for list-ranking is to splice items out of the list until it is reduced to $n/\log n$ algorithm, then apply the basic algorithm, and then reconstruct the list. The splice out phase is precisely what is accomplished by our vertex elimination algorithm. Each processor initially has $\log n$ list cells. When a pair of messages $P_a \rightarrow M_b$, $P_b \rightarrow M_c$ get through, one of the list cells assigned to processor P_b can be spliced out of the list. The algorithm insures that adjacent list cells are not spliced out in the same phase. After the problem is reduced to size $cn/\log n$, it can be solved on $n/\log n$ processors. The problem is then solved by rebuilding the list by essentially running the splice out algorithm in reverse. An interesting feature of our list-ranking algorithm is that it does not rely on list being doubly linked. The other known efficient list-ranking algorithms all require that the list is doubly linked.

There are many problems that are reducible to list-ranking, so that they can also be solved optimally on the local memory model. In particular, tree contraction can be reduced to list-ranking so it can be solved in $O(\log n)$ time with $n/\log n$ processors in this model [MR85,GMT86].

5 Conclusions

In this paper we have investigated algorithms for the local memory P-RAM. The model was motivated by a consideration of potential optical computers, where unit cost communication is possible, but there is a bottleneck in accessing memory modules. Our main result is the introduction of a general approach for algorithms on this type of machine. As a corollary, we give an optimal randomized algorithm for list-ranking. The list-ranking algorithm has the interesting features that it relies on relatively weak synchronization and that it does not require that the list is first doubly linked.

References

[Bel86] T. E. Bell. Optical computing: a field in flux. *IEEE Spectrum*, 34–57, August 1986.

[CR73] S. A. Cook and R. R. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[CV86] R. Cole and U. Vishkin. Approximate scheduling, exact scheduling, and applications to parallel algorithms. In *27th Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[GMT86] H. Gazit, G. L. Miller, and S. H. Teng. Optimal tree contraction in the EREW model. 1986. Extended abstract.

[KU86] A. R. Karlin and E. Upfal. Parallel hashing - an efficient implementation of shared memory. In *Proceedings of the 18th ACM Symposium on Theory of Computation*, pages 160–168, 1986.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its applications. In *26th Symposium on Foundations of Computer Science*, pages 478–489, 1985.

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

[Ran87] A. G. Ranade. How to emulate shared memory. In *28th Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[Rei] J. H. Reif. Personal Communication, October 1987.

[SJRV87] A. A. Sawchuck, B. K. Jenkins, C. S. Raghavendra, and A. Varma. Optical crossbar networks. *Computers*, 20(6):50–60, June 1987.

[Tho80] C. D. Thompson. *A Complexity Theory for VLSI.* PhD thesis, Carnegie-Mellon University, 1980.

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

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computation.* PhD thesis, Department of Computer Science, Cornell University, 1979.