21

The Complexity of Computation on the Parallel Random Access Machine

Faith E. Fich

Department of Computer Science
University of Toronto
10 King's College Road
Toronto, Ontario
CANADA M5S 1A4

This chapter discusses the Parallel Random Access Machine (PRAM), various parameters of this model, under what circumstances these parameters affect its computational power, and how they do so. It also surveys techniques that have been used to obtain lower bounds for specific problems on PRAMs.

21.1 The PRAM Model

The PRAM is a synchronous model of parallel computation in which processors communicate via a shared memory. It consists of m shared memory cells $M_1, \ldots, M_m$ and p processors $P_1, \ldots, P_p$. Each processor is a random access machine (RAM) with a private local memory. During every step of a computation, a processor may read from one shared memory cell, perform a local operation, and write to one shared memory cell. Different processors may execute different operations during a step and may make reference to their own index. Reads, local operations, and writes are viewed as occurring during three separate phases. This simplifies analysis and only changes the running time of an algorithm by a constant factor.

An input x consists of n values $x_1, \ldots, x_n$. At the beginning of the computation, these values are located in shared memory cells $M_1, \ldots, M_n$, respectively, provided $n \leq m$. If there are $n' \leq m$ output values, they appear in the first n' shared memory cells at the end of the computation When the number of shared memory cells is too small, the input values can be distributed approximately equally among the processors. Another possibility is to have a separate, read-only memory containing the input values [VW85, LY89, FLRY89]. In this case, each processor is also allowed to read from one of the n read-only memory cells during each computation step.

EXERCISE 21.1
Explain why it doesn't matter whether the input is initially located in the shared memory, the processors' local memories, or a separate read-only memory, provided the amount of shared memory is large enough to contain the input (i.e. $m \geq n$).

If the amount of shared memory m is small, then where the input is located is more important. For example, determining whether the input contains two consecutive variables with value 1 requires $\Omega(n/m)$ steps when the input is distributed among the processors' local memories (Exercise 21.43).

However, when the input is in a separate read-only memory, this computation can be done significantly faster (Exercise 21.9).

Limiting the amount of shared memory in a PRAM corresponds to restricting the amount of information that can be communicated between processors in one step. For example, a set of processors connected to one another by a single bus may be viewed as a PRAM with one shared memory cell.

THEOREM 21.1

Any problem that can be solved by a PRAM using p processors in t steps can also be solved using $p' \leq p$ processors in $O(tp/p')$ steps.

PROOF

The original p processors are partitioned into p' groups of size at most $\lceil p/p' \rceil$. Each of the p' processors in the simulating machine is associated with one of these groups. To simulate one step of the original computation, each of the p' processors sequentially simulates the read and local computation phases performed by all the processors in its group and then sequentially simulates their write phases. ∎

EXERCISE 21.2

[ACF90] Give an example of a problem that can be solved in one step on a PRAM with p processors, but requires $2\lceil p/p' \rceil - 1$ steps on a PRAM with p' processors for $1 \leq p' < p$.

EXERCISE 21.3

Prove that any problem that can be solved by a PRAM using m shared memory cells and p processors in t steps can also be solved using $m' \leq m$ shared memory cells and $\max\{p, m'\}$ processors in $O(tm/m')$ steps.

EXERCISE 21.4

[Goo89] A *forking PRAM* is a PRAM in which a new processor is created when an existing processor executes a *fork operation*. In addition to creating the new processor, this operation specifies the task that the new processor is to perform (starting at the next time step). Initially there is one processor.

Prove that if a forking PRAM algorithm uses t steps and p processors, then this algorithm can be implemented on a PRAM using $O(t \log p)$ steps and $O(p/\log p)$ processors. Explain explicitly how work is allocated to the PRAM processors.

One consequence of Theorem 21.1 is that the processor-time product $p \cdot t$ of any algorithm solving a problem is at least a constant factor times the sequential time complexity of the problem. Thus, a lower bound on the sequential time complexity gives a lower bound on the number of processors needed for a PRAM to solve the problem in a given amount of time, as well as a lower bound on the amount of time needed for a PRAM to solve the problem using a given number of processors.

A PRAM algorithm is *optimal* if it is not possible to simultaneously improve both the time and the number of processors by more than a constant factor. If a PRAM algorithm solves a problem in t steps using p processors, where $p \cdot t$ is within a constant factor of the sequential time complexity of the problem, then it is called *efficient* [KRS90]. All efficient algorithms are optimal, but not all optimal algorithms are efficient. For many problems, it is interesting and important to find the fastest possible efficient algorithm.

Some people call a parallel algorithm optimal if its processor-time product is within a constant factor of the running time of the fastest known sequential algorithm solving the same problem. Unless there is a matching sequential lower bound for the problem, difficulties may arise with this definition. Specifically, if a faster sequential algorithm is found, suddenly, the parallel algorithm is no longer optimal. Furthermore, this definition of optimality may preclude calling an algorithm optimal even when it is provably impossible to simultaneously improve both the time and the number of processors. For example, PARITY has linear sequential time complexity, but an exponential number of processors are required by any PRAM that computes it in constant time (Theorem 21.33).

The PRAM is a natural generalization of the unit cost RAM [CR73, AHU74], a commonly used model of sequential computation. Numerous parallel algorithms have been designed for the PRAM because it is a simple, precise model in which parallel algorithms can be expressed using a high level language. The PRAM often corresponds to the programmer's view of parallel computation, ignoring lower level architectural details such as memory organization, routing, memory contention, and synchronization. In essence, PRAM programmers do not have to be concerned with how communication is accomplished, but only what is to be communicated where. PRAMs can be simulated by more realistic models [MV84, UW87, Ran87, AHMP87, HP89] and some real parallel computers [Sch80, Smi90]. Thus, programs written for the PRAM can be compiled into programs for these machines. Although some of the assumptions (such as constant time access to shared memory) are not generally valid, the performance of algorithms on the PRAM can be a good predictor of their relative performance on real machines, especially as problem sizes get large. Furthermore, because the PRAM is a powerful model, the lower bounds obtained for it are automatically applicable to a wide variety of less powerful, more realistic machines.

21.2 Restrictions on Access to Shared Memory

An important parameter of the PRAM model is the extent to which concurrent access to shared memory cells is allowed. If at most one processor can read from a single memory cell at a particular step and at most one processor can write to a single memory cell at a particular step, then the PRAM is called *exclusive read exclusive write* (EREW) [LPV81]. In a *concurrent read exclusive write* (CREW) PRAM [FW78], any number of processors can simultaneously read from the same memory cell, but at most one processor can write to each memory cell at a given step. Many algorithms designed for CREW and EREW PRAMs avoid write conflicts by having each processor own one cell to which all its writes are performed. These restricted models are called the *concurrent-read owner-write* (CROW) PRAM [DR86] and the *exclusive-read owner-write* (EROW) PRAM [FW90], respectively.

When concurrent writes are allowed, it is necessary to specify how conflicts are resolved. Three of the most frequently used *concurrent read concurrent write* (CRCW) PRAMs are considered here. A number of others are discussed in the literature. (For example, see [FRW88a, GR90, EG88, HR90].) The COMMON model [Kuc82] requires that all processors simultaneously writing to the same memory cell write a common value. On the ARBITRARY model [Vis83], an arbitrary one of the values written to a memory cell at a given step will appear in the cell. Any algorithm for this model must work regardless of which value is chosen (say, by an adversary) to resolve each write conflict that arises. In the PRIORITY model [Gol82], processors are assigned fixed, distinct priorities. The processor of highest priority among those that simultaneously write to a memory cell succeeds. Without loss of generality, we assume that lower indexed processors have higher priority.

Any algorithm that runs on ARBITRARY will run unchanged on PRIORITY. Thus PRIORITY is at least as powerful as ARBITRARY. Similarly, ARBITRARY is at least as powerful as COMMON, COMMON is at least as powerful as the CREW PRAM, and the CREW PRAM is at least as powerful as the EREW PRAM. Diagrammatically,

PRIORITY $\geq$ ARBITRARY $\geq$ COMMON $\geq$ CREW PRAM $\geq$ EREW PRAM.

It is important to understand the relationships between these models. Specifically, are some of these models strictly more powerful than others? Can additional resources compensate for restricted access to shared memory? Tradeoffs can help a computer architect choose among a number of design alternatives or help a programmer compare algorithms written on different models. A step by step simulation of a more powerful model by a less powerful model is particularly useful because it allows the straightforward transformation of algorithms designed for the former into algorithms that can run on the latter.

TABLE 21.1
Relationships Between PRAM Models
The fourth column contains bounds on the amount of time for the simulating model using the indicated number of processors to simulate one step of the original model with p processors. For the last three rows, the input is initially distributed among the processors' local memories and both the original and simulating models have m shared memory cells.

Original Model	**Simulating Model**	**Number of Processors**	**Amount of Time**	**Reference**
PRIORITY ARBITRARY COMMON	EREW PRAM CREW PRAM	$\geq p$	$\Theta(\log p)$	Thm 21.2 Thm 21.3 Thm 21.23
PRIORITY ARBITRARY	COMMON	kp	$\Theta\left(\frac{\log p}{k(\log\log p - \log p)}\right)$	Thm 21.7 Ex 21.13
PRIORITY	ARBITRARY	kp	$O\left(\frac{\log\log p}{\log(k+1))}\right)$	Ex 21.14
CREW PRAM	EREW PRAM	≥ 1	$\Omega(\sqrt{\log p}/\log\log p)$	Ex 21.39
PRIORITY ARBITRARY COMMON	EREW PRAM CREW PRAM	$\geq p + m^2$	$\Theta(p/m)$	Ex 21.7 Ex 21.44
PRIORITY ARBITRARY	COMMON	p	$\Omega(\log(p/m))$	Thm 21.42
PRIORITY	ARBITRARY	p	$\Omega(\log(p/m))$	Ex 21.41

Table 21.1 summarizes some of the known relationships between the PRAM models discussed above. The remainder of this section discusses these

and related results and the conditions under which they apply. Some of the lower bounds are presented in Section 4.

The first simulation is quite simple. Moreover, it contains ideas that are used in other simulations in this section.

THEOREM 21.2 [FRW88a]

One step of PRIORITY with p processors and m shared memory cells can be simulated by an EREW PRAM in $O(\log p)$ steps with p processors and mp shared memory cells.

PROOF

Each PRIORITY processor is simulated by a corresponding EREW PRAM processor. For each original shared memory cell in PRIORITY, the EREW PRAM will have p shared memory cells. One of these will contain the same sequence of values as the original cell. The other $p-1$ cells are assumed to be initialized to 0 and will be used to resolve conflicting accesses to the cell.

To simulate the write phase, each processor must know whether it is the leftmost processor (i.e. the processor of lowest index) wishing to write into some cell and, if so, it can perform the write. This leads to the definition of the following problem.

> LEFTMOST WRITERS
> Each processor P_i has a value in the range $\{0, \ldots, m\}$ in its local memory, denoting the shared memory cell that P_i wants to access. The value 0 indicates that P_i does not want to access any cell. Each processor P_i must determine whether it is the leftmost processor among those with the same nonzero value.

One way to solve this problem is to solve m simultaneous instances of the following problem, one for each of the original shared memory cells.

> LEFTMOST PRISONER
> Each processor has a bit, known only to itself, which is 1 if the processor wants to access the given memory cell. Each processor with value 1 must determine whether it is the leftmost processor with value 1. Only processors with value 1 can participate in the computation.

The name of this problem comes from imagining the processors to be prisoners who cooperate with one another to determine the lowest numbered occupied prison cell. Note that all m instances can be solved simultaneously because every processor participates in at most one computation.

The additional $p-1$ memory cells associated with each original memory cell are used to solve its LEFTMOST PRISONER problem. They are viewed as representing the internal nodes of a binary tree with p leaves (one corresponding to each processor) and depth $\lceil \log_2 p \rceil$.

All processors with value 1 are considered to be initially located at the corresponding leaves of the tree. The internal nodes of the tree are processed one level at a time, starting from the bottom. Each processor located at the right child of a node in the current level reads the value stored at the left child. When this value is 0, there is no processor located at the node's left child. In this case, the processor proceeds to the node and writes the value 1 there. Simultaneously, each processor located at the left child of a node in the current level proceeds to the node and writes the value 1 there. The processor that reaches the root of the tree is the leftmost processor that wants to access the given shared memory cell. Hence, it can perform the write to the cell. (An example appears in Figure 21.1.) Finally, processors retrace their steps back down the tree, erasing the values they have written at the nodes.

To simulate the read phase, it is sufficient to choose one processor to read each desired memory cell and distribute its contents to all interested processors. This can also be accomplished by performing the LEFTMOST PRISONER algorithm and having the processor that reaches the root read the given shared memory cell. As the processors go back down the tree, this value is written at each internal node that a processor reaches and is then erased. A processor waiting at a node can read the value from its parent at the appropriate time. ∎

A simplified version of this algorithm can be performed on COMMON without using more shared memory than the original machine.

EXERCISE 21.5
[FRW88a] Prove that one step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in $O(\log p)$ steps with p processors and m shared memory cells.

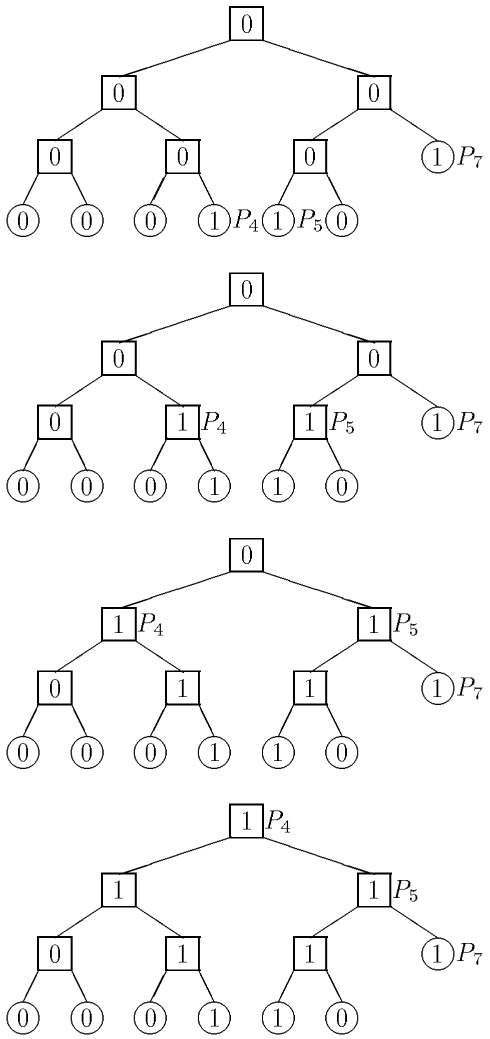

FIGURE 21.1
An example illustrating the processors going up the tree in the LEFTMOST PRISONER algorithm in the proof of Theorem 21.2. Here processors P_4, P_5, and P_7 want to access the shared memory cell and processors P_1, P_2, P_3 and P_6 do not.

The following simulation is more complicated than the simulation in Theorem 21.2, but uses fewer shared memory cells. When $m \in \Omega(p)$, Exercise 21.3 and Theorem 21.1 can be applied to this result to show that no additional processors or shared memory cells are needed for simulating PRIORITY by an EREW PRAM in $O(\log p)$ steps.

THEOREM 21.3 [Eck79, Vis83, NS81]

One step of PRIORITY with p processors and m shared memory cells can be simulated by an EREW PRAM in $O(\log p)$ steps with p processors and $m + p$ shared memory cells.

PROOF

As in the proof of Theorem 21.2, each original shared memory cell has a corresponding shared memory cell in the EREW PRAM and the local operations of each original processor are performed by a corresponding processor in the EREW PRAM. An auxiliary array A of size p is also used.

At a given read or write phase in the PRIORITY algorithm, if processor P_i wants to access cell M_j, it writes the pair (j, i) to memory cell A_i. If P_i doesn't want to access any shared memory cell, it writes $(0, i)$ to A_i. Notice that each of the indices $1, \ldots, p$ occurs as the second component of exactly one element of A. For example, if P_1 wants to access cell M_2, P_2 wants to access cell M_4, P_3 wants to access cell M_2, P_4 wants to access cell M_1, P_5 wants to access cell M_4, P_6 wants to access cell M_2, and P_7 does not want to access any cell, the array A would have the following contents.

(2,1)	(4,2)	(2,3)	(1,4)	(4,5)	(2,6)	(0,7)

The array A is then sorted into lexicographic order. This takes $O(\log p)$ steps on an EREW PRAM with p processors. (See Chapter 12.)

(0,7)	(1,4)	(2,1)	(2,3)	(2,6)	(4,2)	(4,5)

Next, each processor P_i appends a bit to the contents of cell A_i. This bit is 0 when the first component of A_i is either 0 or the same as the first component of A_{i-1}; otherwise this bit is 1. Observe that processor P_i is the highest priority processor wishing to access cell M_j if the triple $(j, i, 1)$ appears in the array A.

(0,7,0)	(1,4,1)	(2,1,1)	(2,3,0)	(2,6,0)	(4,2,1)	(4,5,0)

In the example, P_4 is the highest priority processor that wants to access M_1, P_1 is the highest priority processor that wants to access M_2, and P_2 is the highest priority processor that wants to access M_4.

From this point, it is easy to finish the simulation of a write phase. First, each processor P_k reads the triple (j, i, b) in A_k and writes this triple into A_i. Then processor P_i reads the triple (j, i, b) from A_i. If the bit b has value 1, then P_i is the highest priority processor that wants to write to M_j, so it can perform its write.

(2,1,1)	(4,2,1)	(2,3,0)	(1,4,1)	(4,5,0)	(2,6,0)	(0,7,0)

In this case, P_1 writes to cell M_2, P_2 writes to cell M_4, and P_4 writes to cell M_1.

Once the appropriate bit has been appended to the contents of each cell of A, a read phase can be simulated as follows. Each processor P_k reads the triple (j, i, b) in A_k. If the bit b has value 1, then P_k reads shared memory cell M_j and overwrites the third component of A_k with the value v_j it read. These values are then duplicated as appropriate. Specifically, for $l = \lceil \log_2 p \rceil, \ldots, 1$, processor P_k overwrites the third component of $A_{k+2^{l-1}}$ with the third component of A_k, provided they have the same first components. Here are the successive contents of the array in the example.

(0,7,0)	(1,4,v_1)	(2,1,v_2)	(2,3,0)	(2,6,0)	(4,2,v_4)	(4,5,0)

(0,7,0)	(1,4,v_1)	(2,1,v_2)	(2,3,v_2)	(2,6,0)	(4,2,v_4)	(4,5,v_4)

(0,7,0)	(1,4,v_1)	(2,1,v_2)	(2,3,v_2)	(2,6,v_2)	(4,2,v_4)	(4,5,v_4)

Finally, each processor P_k reads the triple (j, i, b) in A_k and writes it into A_i.

(2,1,v_2)	(4,2,v_4)	(2,3,v_2)	(1,4,v_1)	(4,5,v_4)	(2,6,v_2)	(0,7,0)

Now processor P_i can read the value it needs from A_i. ∎

Concurrent write PRAMs with p processors cannot, in general, be simulated by exclusive write PRAMs without an $\Omega(\log p)$ factor increase in time, even with an infinite number of processors and shared memory cells. This is because the OR of n Boolean variables can be computed in one step on COMMON with n processors and one shared memory cell, but requires $\Omega(\log n)$

steps on a CREW PRAM with an infinite number of processors and shared memory cells. (See Theorem 21.23.)

EXERCISE 21.6
Prove that the OR of n Boolean variables can be computed in $O(\log n)$ steps on an EROW PRAM with $n/\log_2 n$ processors and $n/\log_2 n$ shared memory cells.

When the amount of shared memory m is small, significantly more time is needed to compute the OR of n input bits on a CREW PRAM, even with an infinite number of processors. If the input bits are initially located in the processors' local memories, then $\Omega(n/m)$ steps are required (see Exercise 21.44), whereas, if they are located in a separate read-only shared memory, $\Omega\left(\sqrt{n/m}\right)$ steps are required [Bea87, VW85].

EXERCISE 21.7
Prove that one step of PRIORITY with p processors and m shared memory cells can be simulated by an EREW PRAM in $O(p/m)$ steps with $p + m^2$ processors and m shared memory cells.

EXERCISE 21.8
[VW85] Prove that the OR of n Boolean variables located in a separate read-only shared memory can be computed in $O\left(\sqrt{n/m} + \log m\right)$ steps on an EREW PRAM with $O(\sqrt{nm})$ processors and m shared memory cells.

EXERCISE 21.9
Suppose the input consists of n bits located in a separate read-only shared memory. Prove that, in $O(\sqrt{n/m} + \log m)$ steps, an EREW PRAM with $O(\sqrt{nm})$ processors and m shared memory cells can determine whether the input contains two consecutive variables with value 1. Prove that this problem can be solved in $O(1)$ steps on COMMON with n processors and 1 shared memory cell.

Both COMMON and ARBITRARY can simulate PRIORITY with only a constant factor increase in time, by increasing the number of processors and the amount of shared memory.

THEOREM 21.4 [Kuc82]

One step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in $O(1)$ steps with $\binom{p}{2}$ processors and $m + p$ shared memory cells.

PROOF

Processors $P_1, \ldots, P_p$ are responsible for simulating the original processors. Simulation of the read and compute phases is trivial, since the two models do not differ in this respect. To simulate the write phase, each processor P_i writes the index of the location to which it wishes to write into the auxiliary shared memory cell A_i. As in the proof of Theorem 21.3, the value 0 denotes the fact that P_i does not wish to write during this step. For example, the array A might contain the following values.

2	4	2	1	4	2	0

Each of the $\binom{p}{2}$ processors then reads a different pair of memory cells in the array. If A_i and A_j contain the same value and $i < j$, the processor that read these two cells writes 0 into A_j. This results in the following array.

2	4	0	1	0	0	0

Finally, for $1 \leq i \leq p$, memory cell A_i is read by processor P_i. If A_i contains a nonzero value j, then processor P_i is the lowest indexed processor that wants to write to location M_j and, thus, it can perform the write. ∎

The number of processors used to achieve a constant time simulation can be reduced if more shared memory is available.

EXERCISE 21.10
[FRW88a] Prove that one step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in $O(1/\varepsilon)$ steps with $\min\{p + mp^{\varepsilon}, p^{1+\varepsilon}\}$ processors and $mp^{\varepsilon/2}$ shared memory cells, for any $\varepsilon \in O(1)$.

EXERCISE 21.11
[Cha91] A *semi-oblivious* PRAM algorithm is a PRAM algorithm such that, for each processor and at each step of the computation, there is at most one shared memory cell to which it writes. However, whether a

particular processor decides to write at a particular step may depend on the state it is in (and hence on the values of the input). Prove that any step of a semi-oblivious PRIORITY algorithm that uses p processors and m shared memory cells can be simulated in constant time by a semi-oblivious COMMON algorithm using p processors and $m+p$ shared memory cells.

THEOREM 21.5 [CDHR88]

One step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in $O(1)$ steps with $p \log_2 p$ processors and $mp + p$ shared memory cells.

PROOF

As in the proof of Theorem 21.2, for each shared memory cell M_j, the LEFTMOST PRISONER problem is solved using a binary tree with p leaves. However, instead of having processors proceed up the tree one level at a time, the bit values at all the internal nodes are determined simultaneously.

For every original processor P_i, there are $\log_2 p$ processors in the simulating machine. If processor P_i wants to write to memory cell M_j, these processors simultaneously write the value 1 to each ancestor of the ith leaf that has the ith leaf in its left subtree.

Note that, after this step, an internal node has value 1 if and only if there is a leaf in its left subtree corresponding to an original processor that wants to write to M_j. Thus, processor P_i is the leftmost processor that wants to write to memory cell M_j if and only if every ancestor of the ith leaf that has the ith leaf in its right subtree contains the value 0. This can be determined in constant time by computing the OR of these values using one additional memory cell associated with processor P_i.

Consider the example in Figure 21.2. Since P_3 is in its parent's and great grandparent's left subtree and in its grandparent's right subtree, the OR that is computed has value 0. However, P_4 is in its parent's and grandparent's right subtrees and P_5 is in its great grandparent's right subtree, so the ORs that are computed for both these processors have value 1. ∎

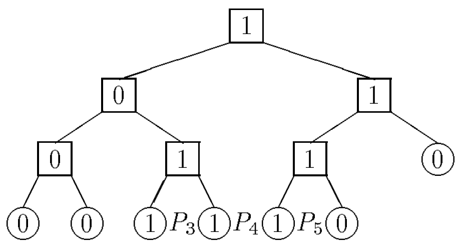

FIGURE 21.2
The result of the first step of the LEFTMOST PRISONER algorithm in the proof of Theorem 21.5. In this example, $p = 7$ and processors P_3, P_4, and P_5 wish to write to the shared memory cell.

Even without increasing the number of processors, it is still possible to simulate PRIORITY faster on COMMON than on the EREW PRAM.

THEOREM 21.6 [FRW88b]

One step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in $O(\log p / \log\log p)$ steps with p processors and $O(mp)$ shared memory cells.

PROOF

Here the LEFTMOST PRISONER problem is solved using a completely different approach. For $p = 2$, the problem can be solved in one write phase and one read phase using one shared memory cell that is initialized to 0. Specifically, if P_1 has value 1, it writes 1 into the cell. In this case, it is the leftmost processor with value 1. If P_2 has value 1, it reads the cell. It is the leftmost processor with value 1 if and only if it reads the value 0.

Given an algorithm that solves the problem for $p = t!$ in $t - 1$ write and read phases using m_{t-1} shared memory cells, we can construct an algorithm that solves the problem for $p = (t+1)!$ in t write and read phases using $m_t = (t+1)m_{t-1} + 1$ shared memory cells, as follows. The problem is divided into $t + 1$ subproblems of size $t!$. Group 1 consists of the first $t!$ processors, group 2 consists of the next $t!$ processors, etc. Each subproblem is solved independently, using m_{t-1} shared memory cells and $t-1$ of the write and read phases. One shared memory cell is used for the groups to interact with one another, so that the group containing the leftmost processor with value 1 can be determined. This cell initially contains the value 0. In the ith write phase, all processors (with value 1) in group i write 1 to the cell and in the next read phase all processors (with value 1) in group $i+1$ read the cell. Notice that after the ith write phase, the cell contains the value 0 if and only if no processors in the first i groups have the value 1. Thus if there are any processors with value 1 in group $i+1$, they will learn whether their group contains the leftmost processor with value 1.

The number of shared memory cells used to solve each leftmost prisoner problem satisfies the recurrence

$$\begin{aligned} m_1 &= 1 \\ m_t &= (t+1)m_{t-1}+1, \text{ for } t>1. \end{aligned}$$

This implies $m_t \in O((t+1)!) = O(p)$. Therefore the total number of shared memory cells used in the simulation is $O(mp)$. ∎

The ideas in the proofs of the two preceding theorems can be combined to get increasingly faster simulations of PRIORITY by COMMON as the number of processors in the simulating machine increases.

EXERCISE 21.12
[Ragb, Bop89] Prove that one step of PRIORITY with p processors and m shared memory cells can be simulated by COMMON in t steps with kp processors, for $1 \leq k \leq \log_2 p$, provided $t \log_2 t \in O(\frac{\log p}{k})$ i.e. $t \in O(\frac{\log p}{k(\log\log p - \log k)})$. How many shared memory cells do you use?

A matching $\Omega(\frac{\log p}{k(\log\log p - \log k)})$ lower bound for solving the LEFTMOST PRISONER problem on COMMON has been achieved [Ragb]. This does not imply that the simulation of PRIORITY by COMMON in Exercise 21.12 is optimal; it only implies that a better simulation cannot be achieved by this approach. However, the algorithm is known to be optimal for simulating either PRIORITY or ARBITRARY with *sufficiently large* amounts of shared memory. Consider the following problem.

ELEMENT DISTINCTNESS
Given n input values $x_1, \ldots, x_n$ in the range $\{1, \ldots, r\}$, determine if they are pairwise distinct (i.e. $x_i \neq x_j$ for $i \neq j$).

ELEMENT DISTINCTNESS is easy to solve on ARBITRARY (and, hence, PRIORITY), given sufficient shared memory.

EXERCISE 21.13
Prove that ELEMENT DISTINCTNESS can be solved in $O(1)$ steps on ARBITRARY using n processors and r shared memory cells.

An algorithm for solving ELEMENT DISTINCTNESS on COMMON can be obtained by simulating the algorithm in Exercise 21.13. using the simulation in Exercise 21.12. If the size, r, of the input domain is sufficiently large, then the resulting algorithm is optimal.

THEOREM 21.7 [Bop89, Edm91]

COMMON with kn processors and an infinite amount of shared memory requires $\Omega(\frac{\log n}{k(\log\log n - \log k)})$ steps to solve ELEMENT DISTINCTNESS, provided the size of the input domain is sufficiently large.

This result implies that, for r sufficiently large, COMMON with kn processors and infinite shared memory requires $\Omega(\frac{\log n}{k(\log\log n - \log k)})$ times as many steps to solve ELEMENT DISTINCTNESS as PRIORITY or ARBITRARY with n processors and r shared memory cells. Thus, the simulation mentioned in Exercise 21.12 is optimal whenever the machine being simulated has a sufficiently large shared memory. However, when the amount of shared memory m is smaller (for example, only exponential in the number of processors), the optimality of this simulation is unknown. The difficulty with using ELEMENT DISTINCTNESS to separate COMMON from ARBITRARY and PRIORITY in this case is that, if the domain size r is small, there are not good lower bounds known for ELEMENT DISTINCTNESS on COMMON and, if r is large, there are not good upper bounds known for ELEMENT DISTINCTNESS on ARBITRARY or PRIORITY.

ARBITRARY can simulate PRIORITY substantially faster than COMMON can.

THEOREM 21.8 [CDHR88]

One step of PRIORITY with p processors and m shared memory cells can be simulated by ARBITRARY in $O(\log\log p)$ steps using p processors and $m(p-1)$ shared memory cells.

PROOF

This algorithm also works by solving the LEFTMOST PRISONER problem. The processors are divided into $\sqrt{p}$ contiguous groups of size $\sqrt{p}$.

Each group chooses a representative processor with value 1, if there is one. This is done as follows, using one cell for each group. The cell is initialized to 0. Then all processors that are in the group and have value 1 attempt to write their indices into the cell. The processor whose index appears in the cell is chosen to be the representative for the group. All processors (with value 1) in the group can then read the cell to find out which processor is the representative.

For example, if there are nine processors, of which P_5, P_6, P_7, and P_9 have value 1, then the memory cells associated with the three groups $\{1,2,3\}$, $\{4,5,6\}$, and $\{7,8,9\}$ could contain the values 0, 6, and 7, respectively.

Determining the leftmost group that contains a processor with value 1 is a subproblem of size $\sqrt{p}$ that is solved recursively by the chosen representatives. Those processors (with value 1) that were not chosen as representatives are used to recursively determine the leftmost processors with value 1 in their groups. Each of these subproblems has size less than $\sqrt{p}$.

The total time to solve a LEFTMOST PRISONER problem of size p satisfies the recurrence $t(p) = t(\sqrt{p}) + O(1)$. This implies $t(p) \in O(\log\log p)$. It can also be shown inductively that at most $p-1$ memory cells are used to solve a LEFTMOST PRISONER problem of size p. ∎

EXERCISE 21.14

[Ragb] Prove that one step of PRIORITY with p processors and m shared memory cells can be simulated by ARBITRARY in $O(\frac{\log\log p}{\log(k+1)})$ steps with kp processors, for $1 \leq k \leq \log_2 p$. How many shared memory cells do you use?

In particular, the result in Exercise 21.14 implies that one step of PRIORITY with p processors can be simulated by ARBITRARY in $O(1/\varepsilon)$ steps using $O(p(\log p)^{\varepsilon})$ processors, for $0 < \varepsilon \leq 1$.

There is a matching lower bound of $\Omega(\frac{\log\log p}{\log(k+1)})$ steps on ARBITRARY for solving a generalized version of the LEFTMOST PRISONER problem in which each input bit is known to a group of k processors [Ragb]. It is open whether a better simulation of PRIORITY by ARBITRARY can be achieved in some other way.

When the amount of shared memory is small and the input is initially distributed among the processors' local memories, $o(\log p)$ time simulations of CRCW PRAMs with stronger write conflict resolution mechanisms by CRCW PRAMs with weaker ones are not possible without increasing the number of processors or shared memory cells. Specifically, $\Omega(\log(p/m))$ steps are required to simulate either one step of ARBITRARY by COMMON or one step of PRIORITY by ARBITRARY, using the same number of processors and the same number of shared memory cells. (See Exercise 21.42 and Theorem 21.41.) If $m \in O(p^{1-\varepsilon})$ for some constant $\varepsilon > 0$, this implies that $\Omega(\log p)$ steps are required.

It is more difficult to obtain lower bounds when the input is located in a separate read-only shared memory. In this case, $\Omega(\log\log(p/m))$ steps are known to be required to simulate ARBITRARY with p processors and m shared memory cells by COMMON with p processors and m shared memory cells and $\Omega(\log(p/m)/\log\log(p/m))$ steps are known to be required to simulate PRIORITY with p processors and m shared memory cells by ARBITRARY with m shared memory cells and any number of processors [FLRY89]. In contrast, PRIORITY and ARBITRARY require the same amount of time to compute any symmetric Boolean function using one shared memory cell [LY86, LY87].

21.3 Relationships Between PRAMs and Other Models

The PRAM is closely related to other models of computation. This section considers relationships between PRAMs and four theoretical models: Boolean circuits, unbounded fan-in Boolean circuits, Turing machines, and decision trees. These relationships are important for obtaining a better understanding of parallel computation and because good upper and lower bounds on PRAMs can often be obtained from corresponding bounds on these models.

A *Boolean circuit* is a directed acyclic graph whose nodes are either *inputs*, which have fan-in 0, or *gates*, which have fan-in at most 2. The inputs are labelled by distinct variables. Each gate is labelled by a Boolean function whose arity is equal to the fan-in of the gate. Some nodes are also designated as *outputs*. The *depth* of a Boolean circuit is the length of the longest directed path from an input to an output and its *size* is the number of gates it contains.

A CROW PRAM can easily simulate a Boolean circuit using one processor for each gate. The processor reads the values at the nodes with edges directed into the gate as they become available, performs the gate computation, and writes the result in its shared memory cell. If the fan-out of every node in the circuit is bounded by a constant, then the simulation can be done using only exclusive reads. However, any Boolean circuit can be converted to a Boolean circuit with nodes of fan-out at most 2 and only constant factor increases in size and depth [HKP]. Therefore concurrent reads are not necessary.

THEOREM 21.9

Any Boolean circuit of depth d and size s can be can be simulated by an EROW PRAM in $O(d)$ steps using s processors and s shared memory cells.

An *unbounded fan-in Boolean circuit* is a directed acyclic graph composed of input nodes with fan-in 0, NOT gates with fan-in 1, and unbounded fan-in AND and OR gates. Its *depth* is the length of the longest directed path from an input to an output and its *size* is the number of edges in the graph. Since COMMON can compute the AND or OR of n bits in one step using n processors and one shared memory cell, it can easily simulate any unbounded fan-in Boolean circuit.

THEOREM 21.10 [SV84]

Any unbounded fan-in Boolean circuit of depth d and size s can be simulated by COMMON in d steps using s processors and s shared memory cells.

Any Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be expressed by a formula of size $O(n2^n)$ in disjunctive or conjunctive normal form and thus can be computed by COMMON in two steps using exponentially many processors and cells. More generally, COMMON can use *table look-up* to compute any function $f : \{0,1\}^n \rightarrow R$ with Boolean inputs in two steps with $n2^n$ processors and 2^n shared memory cells. The idea is to allocate n processors to each of the 2^n possible inputs. Each group checks if it corresponds to the true input by reading the input and performing an AND. One processor from the unique group whose AND is 1 writes the answer. Thus it is impossible to get nontrivial lower bounds on time for CRCW PRAMs without restricting the number of processors or shared memory cells.

EXERCISE 21.15

[CSV84] Prove that any Boolean function computed by a Boolean circuit of depth $O(\log\log n)$ can be computed by an unbounded fan-in circuit of depth two and $n^{O(1)}$ size.

The complexity classes NC^k and AC^k consist of those Boolean functions that can be computed in $O((\log n)^k)$ depth and $n^{O(1)}$ size by Boolean circuits and unbounded fan-in Boolean circuits, respectively. Theorem 21.9 implies that any problem in NC^k can be solved by an EROW PRAM in $O((\log n)^k)$ steps using $n^{O(1)}$ processors. Similarly, by Theorem 21.10, any problem in AC^k can be solved by COMMON in $O((\log n)^k)$ steps using $n^{O(1)}$ processors. It follows from Exercise 21.15 that any problem in NC^k can be computed by an unbounded fan-in circuit of depth $O((\log n)^k / \log\log n)$ and size $n^{O(1)}$. Combined with Theorem 21.10, this result shows that any problem in NC^k can be solved by COMMON in $O((\log n)^k / \log\log n)$ steps using $n^{O(1)}$ processors and $n^{O(1)}$ memory cells. Thus the upper bound on the time for solving problems in NC^k can be improved by allowing concurrent writes.

The *word size* of a PRAM is the maximum number of bits that can be contained in each cell of shared memory. Note that Theorems 21.9 and 21.10 and the corollaries mentioned above remain valid even with the restriction that the word size is 1.

Next, we consider some sequential complexity classes. Specifically, let $DLOG$ denote the class of problems that can be solved by deterministic Turing machines using logarithmic space. Let $NLOG$ be the analogous class for nondeterministic Turing machines. Theorems 21.11 and 21.12 will show that all problems in the classes $DLOG$ and $NLOG$ can be solved in logarithmic time by the EROW PRAM and COMMON, respectively, using a polynomial number of processors.

The *configuration graph* of a Turing machine on a given input has one node for each possible combination of state, head positions, work tape contents, and time. Directed edges correspond to transitions. Note that the configuration graph is acyclic. Furthermore, since a Turing machine has a constant size alphabet, every node has constant fan-in and fan-out.

If a Turing machine uses $O(\log n)$ space, then it uses $n^{O(1)}$ time and its configuration graph has $n^{O(1)}$ nodes. Moreover, the configuration graph can easily be constructed by an EROW PRAM in $O(\log n)$ steps using $n^{O(1)}$ processors. Each node in the configuration graph of a deterministic Turing machine has fan-out at most one. In this case, parallel tree contraction (see Chapter 2) can be used to determine whether there is a path from the initial configuration to an accepting configuration.

THEOREM 21.11 [FW78, KR90]

Any problem in DLOG can be solved by an EROW PRAM in $O(\log n)$ steps using $n^{O(1)}$ processors.

For nondeterministic Turing machines, it suffices to compute the transitive closure of the configuration graph. This can be done by an unbounded fan-in Boolean circuit with $O(\log n)$ depth and $n^{O(1)}$ size [Bor77] and, hence, from Theorem 21.10, by COMMON in $O(\log n)$ steps using $n^{O(1)}$ processors.

THEOREM 21.12

Any problem in NLOG can be solved by COMMON in $O(\log n)$ steps using $n^{O(1)}$ processors.

For any problem in *NP*, a superlogarithmic lower bound on the EREW PRAM with a polynomial number of processors would imply that *DLOG* is strictly contained in *NP*. Similarly, $\omega(\log n/\log\log n)$ lower bounds on COMMON with a polynomial number of processors for problems in *NP* would imply that NC^1 is strictly contained in *NP*. Since it is currently unknown whether the inclusions $NC^1 \subseteq DLOG \subseteq NP$ are strict, such lower bounds could be difficult to obtain.

Unbounded fan-in Boolean circuits can also simulate PRAMs. A *restricted instruction set PRAM* is a PRAM in which only addition, subtraction, comparison, bitwise Boolean operations, read, write, and indirect addressing are allowed. Multiplication and division of small numbers can also be permitted without changing the results.

THEOREM 21.13 [SV84]

A restricted instruction set PRAM with inputs of word size μ using t steps and p processors can be simulated by an unbounded fan-in circuit of depth $O(t)$ and size $(pt\mu)^{O(1)}$.

In particular, any Boolean function that can be computed by a restricted instruction set PRAM in $O((\log n)^k)$ steps using $n^{O(1)}$ processors is in AC^k. If *COMMON*(t) denotes those Boolean functions that can be computed by a restricted instruction set COMMON PRAM in $O(t)$ steps using $n^{O(1)}$ processors, then

$$\begin{array}{ccccccc} AC^0 & \subseteq & NC^1 & \subseteq & DLOG & \subseteq & NLOG \quad \subseteq \\ \| & & \cap & & & & \\ COMMON(1) & & COMMON\left(\frac{\log n}{\log\log n}\right) & & & & \end{array}$$

$$\begin{array}{ccccc} AC^1 & \subseteq \cdots \subseteq & NC^k & \subseteq & AC^k \\ \| & & \cap & & \| \\ COMMON(\log n) & & COMMON\left(\frac{(\log n)^k}{\log\log n}\right) & & COMMON((\log n)^k). \end{array}$$

Note that by Theorem 21.4, these relationships are also valid when COMMON is replaced by ARBITRARY or PRIORITY.

PRAMs that allow a limited number of other instructions such as arbitrary multiplication, division, and shifts have also been studied [Tra88, TLR88].

An easy counting argument [Ruz] shows that almost all Boolean functions require exponential size unbounded fan-in circuits. Combining this result with Theorem 21.13 gives an analogous result for restricted instruction set PRAMs.

COROLLARY 21.14

Almost all Boolean functions require an exponential number of processors to be computed by a restricted instruction set PRAM in polynomial time.

Lower bounds for specific functions on unbounded fan-in Boolean circuits can be translated into lower bounds on restricted instruction set PRAMs. For example, consider the following lower bound for PARITY.

THEOREM 21.15

[Has87] Any unbounded fan-in circuit of depth k that computes PARITY has size $2^{\Omega\left(n^{1/(k-1)}\right)}$.

In particular, any unbounded fan-in circuit with $n^{O(1)}$ size computing the PARITY of n input bits (i.e. $x_1 \oplus \cdots \oplus x_n$) has depth $\Omega(\log n/\log\log n)$. Together with Theorem 21.13, this implies that any restricted instruction set PRAM with $n^{O(1)}$ processors requires $\Omega(\log n/\log\log n)$ time to compute

PARITY. Since PARITY $\in NC^1$, it follows that this lower bound is tight to within a constant factor.

Lower bounds for computing specific problems, such as integer addition, have also been directly obtained on restricted instruction set CRCW PRAMs [MR84]. These proofs and the lower bounds obtained using Theorem 21.13 depend in an essential way on the restricted instruction set.

In contrast, an *abstract* or *ideal* PRAM, places no restriction on the instruction set. At each step, a processor can compute any function of its local information. Furthermore, shared memory cells are allowed to contain arbitrarily large values. Although this model is unrealistic, lower bounds in this model have great generality since they do not depend on any assumptions about the instruction set. These lower bounds are actually lower bounds on the amount of communication, as opposed to the amount of computation, necessary to solve a problem. They give insight into the nature of communication between processors and may point out where bottlenecks in communication arise. Abstract PRAMs also approximate the situation where communication to and from shared memory is much more expensive than local operations, for example, where each processor is located on a separate chip and access to shared memory is through a combining network.

Not surprisingly, abstract PRAMs can be much more powerful than restricted instruction set PRAMs.

THEOREM 21.16

Any function of n variables can be computed by an abstract EROW PRAM in $O(\log n)$ steps using $n/\log_2 n$ processors and $n/2\log_2 n$ shared memory cells.

PROOF

Each processor begins by reading $\log_2 n$ input values and combining them into one large value. The information known by processors are combined in a binary-tree-like fashion. In each round, the remaining processors are grouped into pairs. In each pair, one processor communicates the information it knows about the input to the other processor and then leaves the computation. After $\lceil \log_2 n \rceil$ rounds, one processor knows all n input values. Then this processor computes the answer in a single additional step. ∎

In particular, superlogarithmic lower bounds for time cannot be obtained on an abstract PRAM without severely restricting the numbers of processors or shared memory cells.

EXERCISE 21.16
[Bea87, Bea88] Prove that any function $f : \{0,1\}^n \rightarrow R$ can be computed by abstract COMMON in $\log_2 n - \log_2 \log_2(p/n) + O(1)$ steps using $p \geq 2n$ processors and $p/\log_2(p/n)$ shared memory cells.

Processors in an abstract PRAM can read and write arbitrarily large values. However, to handle large input and output values, unbounded fan-in Boolean circuits need large numbers of gates. Except for this technicality, unbounded fan-in Boolean circuits can simulate abstract PRAMs.

THEOREM 21.17 [LY89]

If an (abstract CRCW) PRAM computes a Boolean function in t steps using p processors, then the function can be computed by an unbounded fan-in Boolean circuit of depth $O(t)$ and size $p^{2^{t+O(1)}}$.

This result was proved using Kolmogorov complexity. Specifically, given a PRAM, the number of bits needed to describe the states of processors and the addresses and contents of accessed shared memory cells does not grow too quickly with time. In constant depth, an unbounded fan-in Boolean circuit can compute these descriptions (rather than the actual states, addresses, and contents) at any given step from the descriptions at the previous step.

COROLLARY 21.18 *[BH89, Bea87]*

Almost all Boolean functions of n variables require $\log_2 n - \log_2 \log_2 p + \Omega(1)$ *steps to be computed by an (abstract CRCW) PRAM with p processors.*

Together, theorems 21.15 and 21.17 imply that a PRAM with $n^{O(1)}$ processors requires $\Omega(\sqrt{\log n})$ steps to compute the PARITY of n bits. Although this lower bound is not tight, the proof is much easier than the direct proof of Theorem 21.33.

Limiting the word size can significantly affect the amount of time required to solve certain problems. Different organizations of memory, such as a small number of shared memory cells with large word size or a large number with small word size, can be better for different problems [Bel88, Bel91]. For reasonable bounds on the word size, improved versions of Theorem 21.17 and Corollary 21.18 can be obtained.

THEOREM 21.19 [Bel88, Bel91]

If an (abstract CRCW) PRAM with word size μ computes a Boolean function using t steps and p processors, then the function can be computed by an unbounded fan-in circuit of depth $O(t)$ and size $p^{O(1)}2^{O(t\mu)}$.

COROLLARY 21.20

Almost all Boolean functions of n variables require $\frac{n-\log p}{\mu}$ steps to be computed by an (abstract CRCW) PRAM with word size μ using p processors.

From Theorems 21.15 and 21.19, it also follows that an (abstract CRCW) PRAM with $(\log n)^{O(1)}$ word size and $n^{(\log n)^{O(1)}}$ processors requires $\Omega(\log n/\log\log n)$ steps to compute the PARITY of n input bits. Since PARITY is in NC^1, this lower bound on time is tight to within a constant factor.

The *decision tree* is a widely used model for obtaining sequential lower bounds. An algorithm consists of a tree in which every internal node is labelled by an input variable and every leaf is labelled by a possible answer. An internal node has one child corresponding to each possible value of the input variable that labels it. The execution of a decision tree algorithm begins at its root. When an internal node is visited, the value of the input variable labelling the node determines the appropriate child to visit next. The output is the label of the leaf that is reached. The *decision tree complexity* of a problem is the minimum depth of any decision tree that solves the problem (expressed as a function of the number of input variables). Like the abstract PRAM, the decision tree ignores the computation used to solve a problem. Rather, it focuses attention on how much of the input must be examined to determine the answer. There is a very close correspondence between the decision tree model and the CROW PRAM.

THEOREM 21.21 [Raga]

Any function that can be computed by a CROW PRAM in t steps can be computed by a decision tree of height 2^t.

PROOF

Consider any CROW PRAM that computes a function in t steps using p processors. For $1 \leq i \leq p$ and $1 \leq t' \leq t$, let $S(i, t')$ denote an ordered pair consisting of the state of processor P_i and the contents of its corresponding memory cell M_i immediately after step t'. We will inductively define decision trees $T(i, t')$ that compute $S(i, t')$. The desired decision tree can then be obtained from $T(1, t)$ if the ordered pair labelling each leaf is replaced by its second component.

Let q_i be the initial state of processor P_i. If M_i initially contains the ith input value, then $T(i, 0)$ is a tree of height 1. Its root is labelled x_i and there is a leaf labelled (q_i, v) for each possible value v of x_i. Otherwise, M_i initially contains the value 0 and $T(i, 0)$ consists of a single node labelled $(q_i, 0)$.

The decision tree $T(i, t' + 1)$ is created by modifying the decision tree $T(i, t')$. Each leaf labelled (q, v) in $T(i, t')$ indicates that processor P_i is in state q and memory cell M_i contains v at the end of step t' on those inputs that lead to the leaf. Suppose that, in this state, P_i reads memory cell $M_{i'}$ during step $t' + 1$. Then each leaf of $T(i, t')$ labelled (q, v) is replaced by the decision tree $T(i', t')$ and the labels of the leaves are changed appropriately. Specifically, the label (q', v') of a leaf in this subtree is changed to (q'', v'') if P_i goes into state q'' and writes the value v'' as a result of reading the value v' from $M_{i'}$.

It is easy to verify that the height of $T(i, t')$ is at most $2^{t'}$. ■

THEOREM 21.22

[Raga] If a function can be computed by a decision tree of height h, then it can be computed by a CROW PRAM in $1 + \lceil \log_2 h \rceil$ steps.

PROOF

Consider any decision tree of height h. Associate one CROW PRAM processor with each node in the decision tree. Let processor P_1 be associated with the root. To begin the computation, each processor associated with an internal node reads the value of the input variable labelling its node and writes a pointer to the processor associated with the child corresponding to that value. Then, using pointer jumping, the path in the decision tree from the root to a leaf can be determined in $\lceil \log_2 h \rceil$ more steps. ∎

From Theorem 21.21, a lower bound of h on a decision tree implies a lower bound of $\lceil \log_2 h \rceil$ on a CROW PRAM. Thus the logarithm of the decision tree complexity of a problem almost exactly characterizes the time it takes to solve the problem on a CROW PRAM.

EXERCISE 21.17
[Sni] Prove that a CROW PRAM requires $\lceil \log_2 n \rceil$ steps to compute the OR of n input bits, even if it is known that all possible inputs contain at most one bit with value 1.

EXERCISE 21.18
[FR90] Prove that a CROW PRAM requires $\lceil \log_2 (n-1) \rceil$ steps to reverse the direction of the links in a singly linked list of length n.

21.4 Lower Bound Techniques

Many of the lower bound proofs created specifically for (abstract) PRAMs look at how processors and memory cells accumulate knowledge about the input as computation proceeds and what kinds of knowledge can be accumulated. They also find ways of measuring that knowledge and then showing that it cannot increase too quickly.

One of the facts that makes proving PRAM lower bounds difficult is that processors can communicate information by deciding not to write to a particular shared memory cell. This is nicely illustrated by the following example in which $x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$ is computed in 2 steps on an EREW PRAM. Note that, by Exercise 21.17, a CROW PRAM requires 3 steps for this computation.

Initially, memory cells M_1, M_2, M_3, M_4, and M_5 contain the input bits x_1, x_2, x_3, x_4, and x_5, respectively. During step 1,

processor P_1 reads M_2 and, if $x_2 = 1$, writes 1 into M_1,
processor P_3 reads M_4 and, if $x_4 = 1$, writes 1 into M_3, and
processor P_5 reads M_5,

resulting in the following memory contents.

$x_1 \vee x_2$	x_2	$x_3 \vee x_4$	x_4	x_5

At step 2, processor P_5 reads M_3. If $x_5 \vee (x_3 \vee x_4) = 1$, then P_5 writes 1 into M_1. At this point, memory cell M_1 contains the value $x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$, as desired. Note that processor P_5 communicates the fact that $x_3 \vee x_4 \vee x_5 = 0$ by not writing into memory cell M_1.

This idea can be generalized to show that the OR of n bits can be computed almost 1.4 times faster on an EREW PRAM than on a CROW PRAM. The tth Fibonacci number, F_t, is defined by the recurrence

$$\begin{aligned} F_0 &= 0, \\ F_1 &= 1, \text{ and} \\ F_t &= F_{t-1} + F_{t-2}, \text{ for } t \geq 2 \end{aligned}$$

and satisfies the inequalities $((1+\sqrt{5})/2)^{t-2} < F_t \leq ((1+\sqrt{5})/2)^{t-1}$ for $t > 0$.

EXERCISE 21.19
[CDR86] Prove that an EREW PRAM can compute the OR of n input bits using n processors and shared memory cells in t steps, provided $F_{2t+1} \geq n$.

In the example above, there is at most one processor that writes into a particular memory cell at a particular time step on any input. This is not the case for all EREW and CREW PRAM algorithms. Depending on the input, different processors may write into a particular cell at a given step. However, the conditions under which the different processors perform these writes are mutually exclusive. The fact that none of these conditions are satisfied by the input is conveyed when no write occurs. Under some circumstances, this could be a lot of information. For example, if the input is known to contain at most one bit with value 1, then the OR of the input bits can be computed in one step on an EREW PRAM. This is accomplished by having each processor read an input bit and, if it reads the value 1, write 1 to memory cell M_1. In contrast, from Exercise 21.17, a CROW PRAM requires $\lceil \log_2 n \rceil$ steps to perform this computation.

21.4.1 Sensitivity, Block Sensitivity, and Degree

The algorithm in Exercise 21.19 is optimal. More generally, it is possible to characterize, to within a small constant factor, the time needed to compute total n-ary functions on a CREW PRAM in terms of certain simple properties.

A function f with domain $D \subseteq D_1 \times \cdots \times D_n$ and range R is *sensitive to the set of coordinates* $S \subseteq \{1, \ldots, n\}$ *on input* $x \in D$ if there exists an input $y \in D$ such that $f(x) \neq f(y)$ and $x_j = y_j$ for all $j \notin S$. The *sensitivity* or *critical complexity* of f is

$$\max_{x \in D} \#\{i | f \text{ is sensitive to } \{i\} \text{ on input } x\}.$$

For example, the OR of n Boolean variables has sensitivity n (consider the all 0 input) as does their PARITY (consider any input). The function that computes the MAXIMUM of any n input values also has sensitivity n.

The sensitivity of a function can be used to obtain a lower bound on the amount of time necessary to compute it on a CREW PRAM. This is done by showing that the number of coordinates affecting the state of a processor or the contents of a memory cell cannot grow too quickly as the computation proceeds. Formally, coordinate i *affects* processor P (or memory cell M) on input $x \in D_1 \times \cdots \times D_n$ at time t if the state of P (or contents of M) immediately after step t is different on the inputs x and y, for some input $y \in D_1 \times \cdots \times D_n$ that is the same as x except in coordinate i. Since the maximum number of coordinates that affect memory cell M_1 at the end of the computation must be at least the sensitivity of the function f, this leads to a lower bound on time.

THEOREM 21.23 [CDR86]

A CREW PRAM requires at least $\log_b(\mathit{sensitivity}(f))$ *steps to compute a function* $f : D_1 \times \cdots \times D_n \to R$, *where* $b = (5 + \sqrt{21})/2$.

PROOF

Let $s(t)$ and $c(t)$ denote the maximum number of coordinates affecting any processor or shared memory cell, respectively, on any input at time t. Then

$$s(0) = 0 \text{ and } c(0) = 1.$$

Suppose processor P reads from memory cell M on input x during step t. Then any coordinate i affecting P on input x at time t either affects P on input x at time $t-1$ or affects M on input x at time $t-1$. To see this, consider any input y that differs from x only in coordinate i and causes P to have a different state immediately after step t. If, on input y, P is in the same state at time $t-1$ as it is on x, then it reads from cell M during step t. Furthermore, if it finds the same value there, then processor P is in the same state immediately after step t on both these inputs, a contradiction. Thus

$$s(t) \leq s(t-1) + c(t-1).$$

Now consider any shared memory cell M on any input x. There are two cases to consider. If some processor P writes to M on input x during step t, then any coordinate affecting M on input x at time t also affects P on input x at time t. There are at most $s(t)$ such coordinates.

Otherwise no processor writes to M on input x during step t. Let i be any coordinate that affects M on input x at time t. If i does not affect M on input x at time $t-1$, there must be an input $y^{(i)}$ that differs from x only in coordinate i and a processor $P^{(i)}$ that writes to M on input $y^{(i)}$ during step t.

Let j be any other coordinate that affects M on input x at time t. If $P^{(i)} \neq P^{(j)}$, then either coordinate j affects $P^{(i)}$ on input $y^{(i)}$ or coordinate i affects $P^{(j)}$ on input $y^{(j)}$ at time t. Otherwise, during step t, processors $P^{(i)}$ and $P^{(j)}$ would both write to M on input z, where

$$z_k = \begin{cases} y_i^{(i)} & \text{if } k = i \\ y_j^{(j)} & \text{if } k = j \\ x_k & \text{otherwise.} \end{cases}$$

Suppose there are exactly v coordinates that affect M on input x at time t, but not at time $t-1$. Consider a graph whose vertices are these v coordinates. In this graph there is an edge from i to j if and only if $P^{(i)} \neq P^{(j)}$ and coordinate j affects $P^{(i)}$ on input $y^{(i)}$ at time t. Since no processor is affected by more than $s(t)$ coordinates at time t on a given input, this graph contains at most $v \cdot s(t)$ edges.

For any processor P, at most $s(t)$ of the v coordinates i are such that $P = P^{(i)}$. This is because these coordinates all affect P on input x at time t. Thus there are at least $v \cdot (v - s(t))$ ordered pairs (i, j) such that $P^{(i)} \neq P^{(j)}$. At least half of these must be edges in the graph. Hence $v \cdot s(t) \geq v \cdot (v - s(t))/2$, which implies that $v \leq 3s(t)$. Since M is affected by at most $c(t-1) + v$ coordinates at time t,

$$c(t) \leq c(t-1) + 3s(t).$$

From the solution of the resulting recurrence, it follows that $s(t), c(t) < b^t$ so at least $\log_b(\mathit{sensitivity}(f))$ steps are required to compute f. ∎

EXERCISE 21.20
[CDR86] Show that the OR of n input bits can be computed in $k + O(\log k)$ steps on a CREW PRAM if the input is known to contain at most k bits with value 1.

EXERCISE 21.21
Where does the proof of Theorem 21.23 break down for computing the OR of n input bits, when the input is known to contain at most one bit with value 1?

EXERCISE 21.22
Prove that a CREW PRAM requires $\Omega(\min\{k, \log n\})$ steps to compute the OR of n input bits, when the input is known to contain at most k bits with value 1.

EXERCISE 21.23
[Bea87] Prove that no coordinate affects more than $(2+\sqrt{3})^t$ processors or memory cells after t steps of an EREW PRAM computing a Boolean function.

The function f *depends on coordinate* i if f is sensitive to $\{i\}$ on some input $x \in D_1 \times \cdots \times D_n$. In other words, there are two inputs x and y, differing only on coordinate i, such that $f(x) \neq f(y)$.

THEOREM 21.24 [Sim82]
Every function $f : \{0,1\}^n \to \{0,1\}$ *that depends on* k *of its coordinates has sensitivity* $\Omega(\log k)$.

COROLLARY 21.25 *[Sim82]*
A CREW PRAM requires $\Omega(\log\log k)$ *steps to compute any function* $f : \{0,1\}^n \to \{0,1\}$ *that depends on* k *of its coordinates.*

EXERCISE 21.24
Consider the Boolean addressing function $f : \{0,1\}^n \to \{0,1\}$ defined by

$$f(x_1, \ldots, x_r, y_0, \ldots, y_{2^r-1}) = y_j,$$

where $j = \sum_{i=1}^{r} x_i 2^{r-i}$ and $n = r + 2^r$. It uses the binary number formed by concatenating the first r bits as an index to select one of the remaining bits. Prove that this Boolean function has sensitivity $r+1$ and depends on all n of its coordinates. How quickly can you compute this function on a CREW PRAM?

EXERCISE 21.25
Show that the addressing function $f : \{0, \ldots, n-1\} \times \{0,1\}^n \to \{0,1\}$ defined by

$$f(i, y_0, \ldots, y_{n-1}) = y_i$$

can be computed by a PRAM in constant time using only one processor. Explain the difference between this result and Exercise 21.24.

A very useful generalization of sensitivity is *block sensitivity* [Nis89]. For any function f with domain $D \subseteq D_1 \times \cdots \times D_n$, it is defined to be

$$\max_{x \in D} \max\{k | f \text{ is sensitive to } k \text{ disjoint subsets of coordinates on input } x\}.$$

Clearly, the sensitivity of a function f is bounded above by its block sensitivity. However, there are functions whose sensitivity is less than their block sensitivity. (See Exercises 21.26 and 21.27.) It is an open question whether the sensitivity of a function is always at least some polynomial of its block sensitivity or whether it can be exponentially smaller.

EXERCISE 21.26
[Nis89, WZ88] Consider the Boolean function of n variables that has value 1 when exactly $\lfloor n/2 \rfloor$ or $\lfloor n/2 \rfloor + 1$ of the input bits have value 1. What is the sensitivity, the block sensitivity, and the decision tree complexity of this function?

EXERCISE 21.27
[Rub] Exhibit a Boolean function of n variables with sensitivity $O(\sqrt{n})$ and block sensitivity $\Omega(n)$.

The lower bound in Theorem 21.23 can be extended to block sensitivity.

THEOREM 21.26 [Nis89]

A CREW PRAM requires at least $\log_b(\textit{block sensitivity}(f))$ *steps to compute a function* $f : D_1 \times \cdots \times D_n \to R$*, where* $b = (5+\sqrt{21})/2$.

PROOF

Suppose the block sensitivity of f is k and, on input x, f is sensitive to the disjoint sets of coordinates $S_1, \ldots, S_k$. For $j = 1, \ldots, k$, let $y^{(j)} \in D_1 \times \cdots \times D_n$ be an input such that $f(y^{(j)}) \neq f(x)$ and $y_i^{(j)} = x_i$ for all $i \notin S_j$. Given $(z_1, \ldots, z_k) \in \{0,1\}^k$, define $g(z_1, \ldots, z_k)$ to be the value of the function f on the input $w = (w_1, \ldots, w_n)$, constructed as follows. If $z_j = 1$, then make w agree with $y^{(j)}$ on all coordinates in the set S_j. If $z_j = 0$, then make w agree with x on all coordinates in the set S_j. On those coordinates not in any of the sets $S_1, \ldots, S_k$, also make w agree with x. In other words,

$$w_i = \begin{cases} y_i^{(j)} & \text{if } i \in S_j \text{ and } z_j = 1 \\ x_i & \text{otherwise.} \end{cases}$$

The function $g : \{0,1\}^k \to R$ has sensitivity k and can be computed by a CREW PRAM at least as quickly as the function f. It follows that any CREW PRAM requires at least $\log_b(k)$ steps to compute f. ∎

A set of coordinates C is a *certificate* for a function f on an input x if $f(x) = f(y)$ for every input y that agrees with x on all coordinates in C. In other words, knowing the values x_i for all $i \in C$ determines the value of $f(x)$. For example, a set of coordinates corresponding to the variables in a minterm or maxterm of a Boolean function is a certificate. The *certificate complexity* or *nondeterministic decision tree complexity* of a function f with domain $D \subseteq D_1 \times \cdots \times D_n$ is

$$\max_{x \in D} \min\{\#C | C \text{ is a certificate for } f \text{ on input } x\}.$$

The block sensitivity, certificate complexity, and decision tree complexity of a function $f : D_1 \times \cdots \times D_n \to R$ are always closely related.

EXERCISE 21.28
Prove that, for any function $f : D_1 \times \cdots \times D_n \to R$, *block sensitivity*$(f)$ $\leq$ *certificate complexity*(f) $\leq$ *decision tree complexity*(f).

THEOREM 21.27 [Nis89]

For any function $f : D_1 \times \cdots \times D_n \to R$, certificate complexity(f) $\leq$ (block sensitivity(f))^2.

PROOF

Suppose the block sensitivity of f is k. For any input x, let $\mathcal{B}$ be a maximal collection of disjoint sets of coordinates such that each $S \in \mathcal{B}$ is a minimal set of coordinates to which f is sensitive on input x. Note that $|\mathcal{B}| \leq k$. Let $C = \cup\{S | S \in \mathcal{B}\}$ be the coordinates that occur in the sets in $\mathcal{B}$.

Then C is a certificate for f on input x. To see why, suppose there was an input y that agreed with x on all coordinates in C such that $f(x) \neq f(y)$. Let C' be the set of coordinates on which y differs from x. Then f is sensitive to C' on input x. Let C'' be a minimal subset of C' to which f is sensitive on input x. Since C and C'' are disjoint, C'' is disjoint from each $S \in \mathcal{B}$. But then $\mathcal{B} \cup \{C''\}$ is a collection of disjoint minimal sets of coordinates to which f is sensitive on input x, contradicting the maximality of $\mathcal{B}$.

To complete the proof, it suffices to show that $|S| \leq k$ for each $S \in \mathcal{B}$, because this implies that $|C| \leq k^2$. Consider any $S \in \mathcal{B}$ and let y be an input that agrees with x on all coordinates not in S and such that $f(x) \neq f(y)$. Then, by the minimality of S, for each $i \in S$, $\{i\}$ is a set of coordinates to which f is sensitive of input y. Clearly $\{\{i\} | i \in S\}$ is a disjoint collection of sets. Since the block sensitivity of f is k, it follows that $|S| \leq k$. ∎

THEOREM 21.28 [BI87, HH86, Tar88]

For any function $f : D_1 \times \cdots \times D_n \to R$, decision tree complexity(f) $\leq$ (certificate complexity(f))^2.

PROOF

First note that if $f(x) \neq f(y)$, then every certificate C for f on input x intersects every certificate C' for f on input y. Otherwise, it would be possible to construct an input z consistent with x at all coordinates in C and consistent with y at all coordinates in C', which would imply that $f(x) = f(z) = f(y)$.

Suppose $0 \leq l \leq k$ and, for some $r \in R$, every input $x \in f^{-1}(r)$ has a certificate of size at most k and every input $y \notin f^{-1}(r)$ has a certificate of size at most l. We prove by induction that *decision tree complexity*(f) $\leq kl$.

If $|R| = 1$, then f is a constant function and *decision tree complexity*$(f) = 0 \leq kl$. Therefore, assume $|R| > 1$.

If f has no input in its domain with value r, consider the function f' that is identical to f except that it does not contain r in its codomain. Since every input has a certificate of size at most l, *decision tree complexity*$(f) =$ *decision tree complexity*$(f') \leq l^2 \leq kl$.

Otherwise, choose an input in $f^{-1}(r)$ and construct a decision tree T whose first k levels are labelled by the variables in a length k certificate for f on that input. For each node v at depth k, consider the restriction f_v of f to the subdomain in which these k variables have the values specified by the path to v. Since every input $x \in f^{-1}(r)$ has a certificate of size at most k and every input $y \notin f^{-1}(r)$ has a certificate of size at most l that contains at least one of these k variables, every input $x' \in f_v^{-1}(r)$ has a certificate of size at most k and every input $y' \notin f_v^{-1}(r)$ has a certificate of size at most $l-1$. Thus f_v has a decision tree of height at most $k(l-1)$. This tree is rooted at node v of T. The resulting decision tree T computes f and has height at most $k + k(l-1) = kl$. ∎

EXERCISE 21.29
[BSVW86] Prove that, for any monotone Boolean function f, *sensitivity*(f) = *certificate complexity*(f).

Both log(*block sensitivity*(f)) and log(*decision tree complexity*(f)) characterize the time to compute a function $f : D_1 \times \cdots \times D_n \rightarrow R$ on a CREW PRAM to within a small constant factor. This follows from Theorems 21.27 and 21.28, the fact that $\log_b$(*block sensitivity*(f)) is a lower bound on the time for a CREW PRAM to compute f (Theorem 21.26), and the fact that $1 + \lceil \log_2(\textit{decision tree complexity}(f)) \rceil$ is an upper bound on the time for a CROW PRAM and, hence, a CREW PRAM to compute f (Theorem 21.22). Moreover, given a CREW PRAM algorithm computing such a function, it is possible to construct a CROW PRAM algorithm to compute the function to within a constant factor as fast. However, the new algorithm is not necessarily a step by step simulation of the original algorithm and it might use exponentially more processors. It is an open question whether such a large blowup in the number of processors is necessary.

For many of these results, it is essential that the domain of the function f is *complete*, i.e. if there are n inputs, then the domain of f can be expressed as the direct product of n sets of values. In other words, the value of any input variable is not constrained by the values of other input variables. The results are not necessarily true when the domain of f is not complete.

For example, the OR of n bits when it is known that at most one bit is 1 can be computed in one step on a CREW PRAM, but the sensitivity of this function is n and it requires $\log_2 n$ steps on a CROW PRAM. (See Exercises 21.20 and 21.17.) Similarly, reversing a singly linked list or a disjoint collection of circular singly linked lists of length n has decision tree complexity $n-1$ (Exercise 21.18), although this can be done in one step on a CREW PRAM [FR90]. In contrast, when reverse pointers are also present, the situation is quite different.

THEOREM 21.29 [FR90]

A CROW PRAM can compute any function of a disjoint collection of circular doubly linked lists to within a constant factor as fast as a CREW PRAM.

Additional work needs to be done to understand what problems can be solved more quickly by a CREW PRAM than by a CROW PRAM.

Lower bounds that exactly match the upper bounds for certain Boolean functions have been obtained by considering yet another property. For any Boolean function $f : \{0,1\}^n \to \{0,1\}$, there is a unique multilinear polynomial

$$P_f(x) = \sum_{I \subseteq \{1,\ldots,n\}} a_I \prod_{i \in I} x_i$$

with integer coefficients that represents f in the sense that $P_f(x) = f(x)$ whenever $x \in \{0,1\}^n$. Moreover, the coefficients a_I have absolute value at most 2^{n-1} [Smo87]. For example, if $f(x_1, \ldots, x_n) = x_1 \wedge \cdots \wedge x_n$, then $P_f(x_1, \ldots, x_n) = x_1 \cdots x_n$ and if $f(x_1, \ldots, x_n) = x_1 \vee \cdots \vee x_n$, then $P_f(x_1, \ldots, x_n) = 1 - (1-x_1) \cdots (1-x_n)$. The *degree* of the Boolean function f is defined to be the degree of the polynomial P_f.

EXERCISE 21.30

[DKR90] Prove the following facts about the degrees of Boolean functions.

1. $\text{degree}(\bar{f}) = \text{degree}(f)$.
2. $\text{degree}(f \wedge g) \leq \text{degree}(f) + \text{degree}(g)$.
3. $\text{degree}(f \vee g) \leq \text{degree}(f) + \text{degree}(g)$.

4. If $f \wedge g \equiv 0$, then degree$(f \vee g) \leq \max\{$degree(f), degree$(g)\}$.

Since the OR of n bits has degree n, the following result shows that the algorithm in Exercise 21.19 is the fastest possible on a CREW PRAM.

THEOREM 21.30 [DKR90]

If a CREW PRAM computes a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ in t steps, then $F_{2t+1} \geq degree(f)$.

PROOF

Without loss of generality, we make the following two assumptions. First, a processor's state is merely (an encoding of) the sequence of values it has read at each step (so a processor never forgets information). Second, whenever a processor writes, it identifies itself and its current state (i.e. it communicates everything it knows).

For each processor, partition the set of inputs $\{0,1\}^n$ so that inputs are in the same block if and only if they cause the processor to be in the same state immediately after step t. Let $s(t)$ denote the maximum degree of the characteristic function of any block of one of these partitions. Similarly, let $c(t)$ denote the maximum degree of the characteristic function of any block of the partition of $\{0,1\}^n$ induced by a shared memory cell's contents immediately after step t. Then, as in the proof of Theorem 21.23,

$$\begin{aligned} s(0) &= 0, \\ c(0) &= 1, \text{ and} \\ s(t) &\leq s(t-1) + c(t-1), \text{ for } t > 0. \end{aligned}$$

Now consider the characteristic function of any block of the partition induced by the contents of a shared memory cell M, immediately after step t. If, in state q, processor P_i writes into M during step t, then the characteristic function $g_{i,q} : \{0,1\}^n \rightarrow \{0,1\}$ of the set of inputs that cause P_i to be in state q at time t is also the characteristic function of the set of inputs for which M contains the value (i,q) immediately after step t. This function has degree at most $s(t)$, by definition.

Suppose, instead, that no processors write to M during step t and M contains the value v immediately after step t. Let g be the characteristic function identifying those inputs for which M contains the value v

immediately after step $t-1$. Then the characteristic function g' of the corresponding block of the partition induced by M's contents at time t can be expressed as

$$g' = g \wedge \overline{\bigvee_{i,q} g_{i,q}},$$

where the OR is taken over all values of i and q such that, in state q, processor P_i writes to M during step t. Since concurrent writes to M cannot occur, at most one of the functions $g_{i,q}$ has value 1 for a given input. From Exercise 21.30, it follows that $degree\left(\overline{\bigvee_{i,q} g_{i,q}}\right) \leq s(t)$. By definition, $degree(g) \leq c(t-1)$. Thus $degree(g') \leq c(t-1) + s(t)$ and, hence,

$$c(t) \leq c(t-1) + s(t) \text{ for } t > 0.$$

It is easy to prove by induction that $s(t) \leq F_{2t}$ and $c(t) \leq F_{2t+1}$ for all $t \geq 0$. The theorem then follows from the fact that $degree(f)$ is bounded above by the maximum degree of the characteristic functions of the blocks of the partition induced by M_1's contents at the end of the computation. ∎

EXERCISE 21.31
[DKR90] Let $n = k^2$ and consider the function

$$f(x_1, \ldots, x_n) = (x_1 \wedge \cdots \wedge x_k) \vee \cdots \vee (x_{n-k+1} \wedge \cdots \wedge x_n).$$

Prove that the certificate complexity of f is k and its degree is n. What is its decision tree complexity?

EXERCISE 21.32
Prove that the degree of any Boolean function is always bounded above by its decision tree complexity.

THEOREM 21.31 [Sze89]
The block sensitivity of any Boolean function is bounded above by the square of its degree.

For special classes of Boolean functions, better results can be shown.

EXERCISE 21.33
[DKR90] Prove that the sensitivity of a monotone Boolean function is bounded above by its degree.

THEOREM 21.32 [DKR90]

Any nonconstant, symmetric, Boolean function of n variables has degree larger than $n/2$.

The time taken by a randomized CREW PRAM to compute a Boolean function is also related to the function's block sensitivity and degree [Nis89, DKR90]. These results imply that a randomized CREW PRAM cannot compute a Boolean function more than a constant factor faster than a deterministic CROW PRAM.

21.4.2 Simplifying the Algorithmic Structure by Restricting the Input

A variety of lower bounds for the PRAM have been obtained as follows. Given an algorithm to solve a problem, find a restricted problem, either on a smaller number of input variables or a smaller input domain, on which the algorithm behaves in a considerably simpler manner. Then lower bounds are proved directly for this class of simpler algorithms.

It is useful to consider the partitions of the set of inputs into blocks that are indistinguishable to a processor or a memory cell during the first t steps of the computation. As time increases, these partitions become more complicated. The proofs of the next results use the technique of random restrictions to show that these partitions do not become sufficiently complex too quickly. Essentially, after each step of the computation, the values of relatively few randomly chosen bits are fixed. This leaves only a slightly smaller instance of the problem. It can be shown that the resulting partitions of the set of inputs are likely to remain quite simple in structure.

THEOREM 21.33 [BH89]

If PRIORITY computes the PARITY of n input bits in t steps, then it uses $2^{\Omega(n^{1/t})}$ processors and $2^{\Omega((n/t!)^{1/t})}$ shared memory cells.

COROLLARY 21.34 *[BH89]*

PRIORITY with $n^{0(1)}$ processors or shared memory cells requires $\Theta(\log n/\log\log n)$ *steps to compute the PARITY of n input bits.*

THEOREM 21.35 [BH89]

For all $t \in \frac{1}{3}\log n/\log\log n - \omega(\log n/(\log\log n)^2)$, there is a Boolean function of n variables that can be computed by COMMON in t steps using n processors and memory cells, but cannot be computed by PRIORITY in $t-1$ steps using $n^{0(1)}$ processors or using $n^{O(1)}$ memory cells.

Thus, even one extra time step can be more useful than increasing the number of processors or shared memory cells by a polynomial factor or using a more powerful write resolution rule.

An early example of this approach was applied to the following problem [Sni85].

SEARCH AN ORDERED LIST
Given $x_1, \ldots, x_n, y \in \{1, \ldots, r\}$ such that $x_1 \leq x_2 \leq \ldots \leq x_n$, determine that either $y < x_1$ or $x_n \leq y$ or find the index i such that $x_i \leq y < x_{i+1}$.

Using $(p+1)$-ary search, a CREW PRAM with p processors can solve this problem in time $O(\log n/\log(p+1))$. A nontrivial lower bound can be obtained on an EREW PRAM, even for the restricted version of the problem in which $x_1, \ldots, x_n, y \in \{0, 1\}$, by bounding the number of variables affecting processors and memory cells as a function of time. This lower bound remains valid even if concurrent writes are allowed.

EXERCISE 21.34
[Sni85] Prove that an EREW PRAM with p processors requires $\Omega(\log n - \log p)$ steps to SEARCH AN ORDERED LIST of length n.

EXERCISE 21.35
[Sni85] Prove that an EREW PRAM can SEARCH AN ORDERED LIST of length n in $O(\log n - \log p)$ steps using p processors, provided p copies of y are given as part of the input.

EXERCISE 21.36

[Sni85] Prove that an EREW PRAM can SEARCH AN ORDERED LIST of length n in $O(\sqrt{\log n})$ steps using n processors and memory cells.

Even with an arbitrarily large number of processors, the upper bound in Exercise 21.36 cannot be improved. This is a consequence of the limited ability of the EREW PRAM to access the input variable y.

THEOREM 21.36 [Sni85]

An EREW PRAM requires $\Omega(\sqrt{\log n})$ steps to SEARCH AN ORDERED list of length n, provided the domain size, r, is sufficiently large.

The idea of the proof is to show that any algorithm has a simple structure for a large subset of the inputs. Then a lower bound is obtained assuming this simple structure.

Two inputs $x, x' \in D^n$ are *order equivalent* if, for all $i, j \in \{1, \ldots, n\}$,

$$x_i < x_j \text{ if and only if } x'_i < x'_j.$$

A function $f : D^n \to R$ *depends only on the relative values of its variables* if $f(x) = f(x')$ for all order equivalent inputs $x, x' \in D^n$. The *address functions* of a PRAM algorithm are the functions of the input that describe where in shared memory each processor reads from and writes to at each step in the computation. A PRAM algorithm *depends only on the relative values of its variables* if its address functions depend only on the relative values of their variables.

LEMMA 21.37

Consider any EREW PRAM algorithm to SEARCH AN ORDERED LIST. If the domain size r is sufficiently large (as a function of the number of processors, memory cells, and time steps), then there is a large subset $S \subseteq \{1, \ldots, r\}$ such that the EREW PRAM algorithm depends only on the relative values of its variables when restricted to inputs $x_1, \ldots, x_n, y \in S$.

This lemma can be proved by applying the following result from Ramsey theory to each of the address functions.

THEOREM 21.38 [GRS80]

Given a function $f : D^n \to R$, where $|D|$ is sufficiently large in terms of n, $|R|$, and s, there is a subset $S \subseteq D$, with $|S| \geq s$, such that $f|_{S^n}$ depends only on the relative values of its variables.

Finally, any EREW PRAM algorithm to SEARCH AN ORDERED LIST that depends only on the relative values of its variables can be shown to require $\Omega(\sqrt{\log n})$ steps, using arguments similar to those needed for Exercise 21.34 [Sni85].

Furthermore, if the EREW PRAM algorithm stores at most a constant number of input values in each shared memory cell, then it requires $\Theta(\log n/\log\log n)$ steps to SEARCH AN ORDERED LIST. If, in addition, each processor only stores a constant number of input values in its local memory, $\Theta(\log n)$ steps are necessary.

These results demonstrate that concurrent read can be more powerful than exclusive read (because a CREW PRAM with n processors can SEARCH AN ORDERED LIST of n numbers in $O(1)$ steps), although not necessarily for problems with complete domains. However, the lower bounds in Exercise 21.34 and Theorem 21.36 also apply to any problem that has SEARCH AN ORDERED LIST as a special case. Unfortunately, the natural extension to a complete domain, the problem of searching an unordered list, is also difficult for a CREW PRAM.

EXERCISE 21.37

Prove that a CREW PRAM requires $\Omega(\log n)$ time to search an unordered list $x_1, \ldots, x_n$ for an element y.

It is possible to construct a problem with complete domain that has SEARCH AN ORDERED LIST as a special case and can be computed substantially more quickly by a CREW PRAM than by an EREW PRAM [GNR89]. A *comparison tree* is a binary decision tree in which each internal node is labelled by a comparison between two input variables instead of by a single input variable. The two children of an internal node correspond to the outcomes $<$ and $\geq$ of the comparison. As in a decision tree, computation begins at the root and the output is the label of the leaf that is reached. Consider the following comparison tree with n internal nodes and depth $\lceil \log_2 n \rceil$, an example of which is illustrated in Figure 21.3. At the ith internal node (encountered in an inorder traversal), the input variables y and x_i are compared. The leaves are labelled sequentially from left to right with the numbers $0, 1, \ldots, n$. The problem is to determine the output of this comparison tree, given the input $x_1, \ldots, x_n, y \in \{1, \ldots, r\}$. When the input variables $x_1, \ldots, x_n$ are restricted to be in sorted order, this problem is equivalent to SEARCH AN ORDERED LIST.

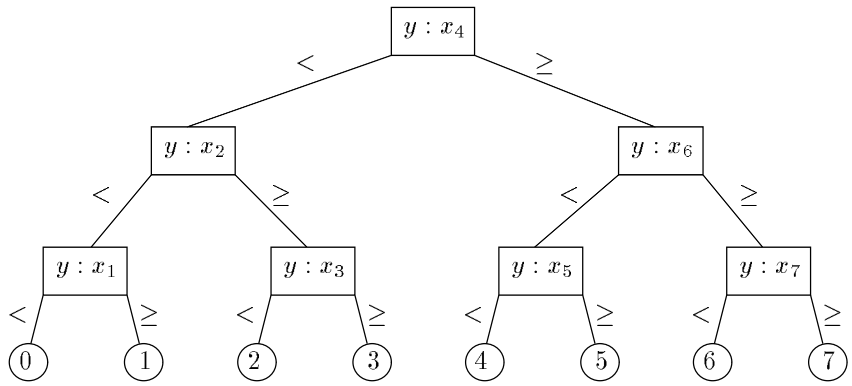

FIGURE 21.3
The comparison tree problem, for $n = 7$.

EXERCISE 21.38
[GNR89] Prove that on a CREW PRAM, the comparison problem can be solved in $O(\log\log n)$ steps, using n processors, and requires $\Omega(\log\log n)$ steps, regardless of the number of processors.

EXERCISE 21.39
Prove that an exclusive read PRAM with an infinite number of processors requires $\Omega(\sqrt{\log p}/\log\log p)$ steps to simulate a general step of a concurrent read PRAM with p processors, when the input domain size is sufficiently large.

EXERCISE 21.40
[GNR89] Construct a function with domain $\{0,1\}^n$ and range $\{0,1\}$ that can be solved in $O(\log\log n)$ steps on a CREW PRAM using n processors, but requires $\Omega(\log n - \log p)$ steps to be computed by an EREW PRAM with p processors.

Lower bounds for a number of other problems have been obtained using the Ramsey theory technique. The ELEMENT DISTINCTNESS problem was studied by Fich, Meyer auf der Heide, and Wigderson [FMW87]. They showed that COMMON with n processors and an infinite amount of shared memory requires $\Omega(\log\log\log n)$ steps to solve this problem for a domain of size $2^{\Omega(n\log n)}$. Ragde, Steiger, Szemeredi, and Wigderson [RSSW88] improved the lower bound on time to $\Omega(\sqrt{\log n})$ and Boppana [Bop89] improved it further to $\Omega(\log n/\log\log n)$, matching the upper bound. (See Theorem 21.7 and Exercise 21.13.) Both results require substantially larger domains. Recently, Edmonds [Edm91] was able to obtain an $\Omega(\log n/\log\log n)$ lower bound using a domain only doubly exponential in n. Boppana [Bop89] and Edmonds [Edm91] also proved that PRIORITY with n processors requires $\Omega(\log n/\log\log n)$ steps to solve ELEMENT DISTINCTNESS on these domains, when the amount of shared memory does not increase as a function of the domain size r. On PRIORITY with n processors (or even $n(\log n)^{O(1)}$ processors) and infinite memory, Fich, Meyer auf der Heide, and Wigderson [MW87] showed that finding the MAXIMUM of n elements requires $\Omega(\log\log n)$ steps (matching Shiloach and Vishkin's upper bound [SV81]) and Meyer auf der Heide and Wigderson [MW87] showed that SORTING a list of length n requires $\Omega(\sqrt{\log n})$ steps. Schieber and Vishkin [SV90] used similar ideas to obtain a tight $\Omega(\log\log n)$ lower bound on the number of steps needed by PRIORITY with $n(\log n)^{0(1)}$ processors and infinite memory to merge two sorted lists of length n or find the nearest neighbour of each vertex in an n-vertex convex polygon.

A serious limitation of these lower bounds, and one that is inherent in the use of Ramsey theory, is that they are only applicable when the size of the problem domain is very large. For example, provided the domain size $r \in O(m)$, ARBITRARY can solve ELEMENT DISTINCTNESS in $O(1)$ steps using n processors and m shared memory cells. (See Exercise 21.13.) Berkman and Vishkin [BV89] showed that two sorted lists of length n containing numbers in the range $\{1,\ldots,r\}$ can be merged on a CREW PRAM in $O(\log\log\log r)$ steps using $n/\log\log\log r$ processors.

EXERCISE 21.41
[FRW88a, EG88] Prove that the MAXIMUM of n elements with values in the range $\{1,\ldots,n^{O(1)}\}$ can be found on COMMON using n processors and memory cells and $O(1)$ time.

At present, it is unknown whether an EREW PRAM can compute every Boolean function as quickly as a CREW PRAM can. However, the following Boolean function require substantially more time to compute on an EROW

PRAM than on a CROW PRAM [FW90].

> BOOLEAN DECISION TREE EVALUATION
> Given the values of 2^m Boolean variables $x_0, \ldots, x_{2^m-1}$ and (the binary encoding of) a complete decision tree of height h, in which each internal node is labelled by one of these 2^m variables and the leaves are alternately labelled 0 and 1, determine the label of the leaf that is reached. The size of the input is $n = 2^m + m(2^h - 1)$.

For example, when $m = 2$ and $h = 3$, the first four bits of the input 111000100111100110 represent the values $x_0 = 1$, $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$. The remaining bits represent the indices of the labels of the interior nodes of the decision tree illustrated in Figure 21.4, when these nodes are arranged according to the inorder traversal of the tree.

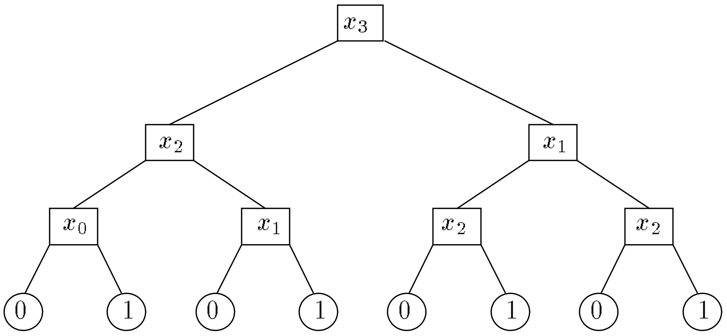

FIGURE 21.4
The Boolean decision tree of height 3 represented by the encoding 00100111100110.

THEOREM 21.39 [FW90]

For an appropriate choice of h and m, a CROW PRAM can solve the BOOLEAN DECISION TREE EVALUATION problem in $O(\log\log n)$ steps, but any (probabilistic) EROW PRAM that computes it requires (expected) $\Omega(\sqrt{\log n})$ steps.

From Theorem 21.22, it follows that a CROW PRAM can solve the BOOLEAN DECISION TREE EVALUATION problem in time $O(\log m + \log h)$. To prove the lower bound, it is sufficient to consider the behaviour of an EROW PRAM algorithm on a randomly chosen input from $\{0,1\}^n$. After each step of the EROW PRAM computation, a longer initial segment of the root to leaf path is revealed. It is very unlikely that any processor knows *both* the value of a variable that has not been revealed and (any bit of) the label of a node in the subtree rooted at the end of this initial segment. In particular, for sufficiently short computations, if a processor knows anything about which variable labels the parent of the leaf that is reached, then it is unlikely to know anything about the value of that variable. It is conjectured that an EREW PRAM requires $(\log n)^{\Omega(1)}$ steps to solve the BOOLEAN DECISION TREE EVALUATION problem, for appropriate h and m.

21.4.3 Small Memory

There are certain techniques that are particularly useful for proving lower bounds on a PRAM with a small amount of shared memory. When a PRAM has only one shared memory cell, we may assume that all processors read the shared memory cell at every step of the computation. The sequence of contents of the m shared memory cells is called the *history* of the computation. Then a processor's knowledge can be expressed as a function of the history and any input values initially located in its local memory. A lower bound of $\Omega(t/m)$ steps on a PRAM using m shared memory cells can be obtained from a lower bound of t steps on a PRAM using one shared memory cell by applying the result of Exercise 21.3.

The following information theoretic lower bound is obtained by considering the partition of the set of all possible inputs into blocks that cause the shared memory to have exactly the same history up to a given point in time. Since two processors may not attempt to write different values in the same memory cell at the same time, it must be possible to infer some "mutual exclusion" information from the history. This mutual exclusion information takes time to set up and is not reusable, so the partition cannot become too fine too quickly.

THEOREM 21.40 [FRW88a]

(Probabilistic) COMMON with n processors and m shared memory cells that computes a surjective function $f : \{0,1\}^n \to R$ requires (expected time) $\Omega((\log |R|)/m)$ steps for some input.

EXERCISE 21.42

[FRW88a] Prove that COMMON with p processors and m shared memory cells requires $\Omega(\log(p/m))$ steps to simulate one step of ARBITRARY or PRIORITY with p processors and m memory cells.

Adversary arguments are also useful for proving lower bounds with small amounts of shared memory. As the computation proceeds, the adversary fixes the history by fixing the values of some of the input variables and, in the case of ARBITRARY, the outcomes of write conflicts. Unless the computation is sufficiently long, this information is insufficient to determine the answer.

THEOREM 21.41 [FRW88a]

ARBITRARY with p processors and m shared memory cells requires $\Omega(\log(p/m))$ *steps to simulate one step of PRIORITY with p processors and m memory cells.*

PROOF

It suffices to prove the lower bound for the following restricted version of the LEFTMOST WRITERS problem. The p processors are divided into m groups of approximately equal size and, for $i \in \{1, \ldots, m\}$, each processor in group i must have value 0 or i. Consider any ARBITRARY algorithm that solves this problem.

A processor whose value has not yet been fixed by the adversary is said to be *free*. The adversary fixes the values of certain processors after each step of the computation in a way that allows any free processor to be the leftmost writer in its group. This is accomplished by never fixing the value of a processor to anything but 0 if there is a free processor of lower index within the same group. As long as there is at least one free processor, the algorithm cannot have terminated.

Initially, the rightmost processor in each group has its value fixed to the index of the group. The values of the other processors are free. Once the history for the first t steps has been fixed, the action of each processor at step $t+1$ can be viewed as a function of its value. Based on these actions, the adversary fixes the contents of each of the m shared memory cells at step $t+1$, as follows.

If possible, the adversary fixes the contents of a cell by choosing the value written at step $t+1$ by a processor whose value has already been fixed. Otherwise, if there is a free processor that writes to the cell at step $t+1$ when its value is 0, the adversary fixes the value of that processor to 0 and fixes the contents of the cell by choosing the value written by that processor.

The remaining unfixed cells are only written to by free processors when their values are nonzero. If possible, the adversary fixes the contents of such a cell by choosing the value written by a processor among the right half of the free processors in some group. The value of the processor is set to the index of its group to ensure that the write takes place. The value of every free processor of higher index within the same group is fixed to 0.

Note that each time the contents of a cell is fixed in one of these ways, the number of possible answers decreases by at most a factor of 2.

Finally, the left half of the free processors in each group have their values fixed to 0, ensuring that no processors write to the remaining unfixed cells. The adversary fixes the contents of those cells to be the same as at step t. This decreases the number of possible answers by a factor of 2^m.

Initially, there are $\Omega((n/m)^m)$ possible answers. This is because any processor in group i can be the leftmost writer to cell i. At each step, the total number of possible answers decreases by a factor of at most 2^{2m}. Thus any algorithm must perform $\Omega(\log(n/m))$ steps to determine the answer in the worst case. ∎

EXERCISE 21.43
[FLRY89] Prove that PRIORITY with an infinite number of processors and m shared memory cells requires $\Omega(n/m)$ steps to determine whether the input $x_1, \ldots, x_n$ contains two consecutive variables with value 1, assuming that the input is initially located in the processors' local memories.

EXERCISE 21.44
Prove that a CREW PRAM with an infinite number of processors and m shared memory cells requires $\Omega(n/m)$ steps to compute the OR of n input bits that are initially located in the processors' local memories.

21.4.4 Reductions

The most frequently used lower bound technique is to find a problem for a which a good lower bound is known and reduce it to the problem of interest. Specifically, let f be a problem with n input variables and let g be a problem with n' input variables. Suppose a PRAM can map each instance $(x_1, \ldots, x_n)$ of f to an instance $(y_1, \ldots, y_{n'})$ of g in t steps and can map the answer $g(y_1, \ldots, y_{n'})$ to the answer $f(x_1, \ldots, x_n)$ in t' steps. If the PRAM requires at least T steps to solve f, then it requires at least $T - t - t'$ steps to solve g. To get a nontrivial lower bound for g, it is important that the time $t + t'$ used to perform the reduction is significantly less than the lower bound T for f. For example, the problem of merging two sorted lists of length n can be reduced in constant time to the problem of triangulating a monotone polygon with $\Theta(n)$ vertices by an EREW PRAM using $O(n)$ processors [BSV88]. Since the former problem requires $\Omega(\log\log n)$ steps on PRIORITY with $n(\log n)^{O(1)}$ processors (see Section 21.21), it follows that this lower bound also applies to the latter problem. More generally, if merging two sorted lists of length n can be reduced to a problem with $O(n)$ inputs on PRIORITY using $o(\log\log n)$ steps and $n(\log n)^{O(1)}$ processors, then that problem requires $\Omega(\log\log n)$ steps on PRIORITY with $n(\log n)^{O(1)}$ processors.

Sometimes it is possible to express each component of g's input as a function of at most one component of f's input and each component of f's answer as a function of at most one component of g's answer. Then f can be reduced to g by a CREW PRAM in constant time using one processor for each component of g's input and one processor for each component of f's answer. This type of reduction is called a *projection.*

THEOREM 21.42

The PARITY of n bits can be reduced via a projection to LIST RANKING in a list of length $3n + 1$.

PROOF

Consider an instance $(x_1, \ldots, x_n)$ of the PARITY problem. Create an instance of LIST RANKING with nodes $1, \ldots, 2n + 1$ arranged in order starting at 1. If $x_i = 0$, then node $2n + i + 1$ is placed between nodes $n + i$ and $n + i + 1$ and if $x_i = 1$, then it is placed between nodes i and $i + 1$. In particular, the distance from node $n + 1$ to the end of the list is the number of input variables with value 1 plus twice the number of input variables with value 0. Hence, the least significant bit of the rank of node $n + 1$ is the PARITY of $x_1, \ldots, x_n$. ∎

From Theorems 21.23 and 21.33, it follows that LIST RANKING requires $\Omega(\log n)$ steps on a CREW PRAM using any number of processors and $\Omega(\log n/\log\log n)$ steps on PRIORITY using a polynomial number of processors.

EXERCISE 21.45
[FSS84] Prove that there is a projection mapping the PARITY of n bits to the MULTIPLICATION of two n bit numbers.

Many other examples of projections appear in [SV85] and [CSV84].

An analogue of Turing reducibility is also useful for proving lower bounds. Let f be a problem with n inputs and let $\{g_i\}$ be a family of problems where g_i has i inputs. Suppose there is a PRAM that can solve f using t time steps and p processors, given access to an oracle that solves any instance of g_i for $i \leq s(n)$. If the PRAM can solve g_i using $t'(i)$ time steps and $p'(i)$ processors, then it can solve f using at most $t \cdot t'(s(n))$ time steps and $p \cdot p'(s(n))$ processors. Furthermore, if no input to an oracle computation is a function of an output of another oracle computation, then the PRAM can solve f using only $t + t'(s(n))$ time steps.

A special case of such a reduction is one that can be performed by a constant depth, polynomial size unbounded fan-in Boolean circuit with gates that compute any bit of the output of a problem in $\{g_i\}$. This reduction can also be performed by COMMON (and, hence, PRIORITY) in constant time using a polynomial number of processors. (See Theorem 21.10.) Constant depth, polynomial size reductions from determining the PARITY of n bits to ADDING, SORTING, and determining the MAJORITY of $O(n)$ bits and computing the TRANSITIVE CLOSURE of an $n+2$ node graph [FSS84, CSV84] imply $\Omega(\log n/\log\log n)$ lower bounds for these problems on PRIORITY with $n^{0(1)}$ processors. Other examples of constant depth, polynomial size reductions appear in [FSS84, CSV84, ACG$^+$88].

EXERCISE 21.46
[CSV84] Prove that SORTING n n-bit integers is reducible to determining (the bits of) the binary representation of the sum of n bits and vice versa.

Acknowledgements

I am grateful to Jeff Edmonds, David Neto, Naomi Nishimura, Prabhakar Ragde, and Jeannine St. Jacques for carefully reading various drafts of this chapter and making valuable comments. Preparation of this chapter was supported by the Natural Sciences and Engineering Research Council of Canada (grant A9176) and the Information Technology Research Centre of Ontario.

Bibliography

[ACF90] S. Akl, M. Cosnard, and A. Ferreira. Data-movement-intensive problems: Two folk theorems in parallel computation revisited. Technical Report 90-18, Ecole Normale Supérieure de Lyon, June 1990.

[ACG+88] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3:293–327, 1988.

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

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

[Bea87] P. Beame. *Lower Bounds in Parallel Machine Computation*. PhD thesis, University of Toronto, 1987. Department of Computer Science Tech. Report 198/87.

[Bea88] P. Beame. Limits on the power of concurrent-write parallel machines. *Information and Computation*, 76(1):13–28, 1988.

[Bel88] S. Bellantoni. Parallel RAMs with bounded memory wordsize. Master's thesis, University of Toronto, 1988.

[Bel91] S. Bellantoni. Parallel RAMs with bounded memory wordsize. *Information and Computation*, 91(2):259–273, April 1991.

[BH89] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, 36(3):643–670, 1989.

[BI87] M. Blum and R. Impagliazzo. Generic oracles and oracle classes. In *FOCS*, pages 118–126, 1987.

[Bop89] R. Boppana. Optimal separations between concurrent-write parallel machines. In *STOC*, pages 320–326, 1989.

[Bor77] A. Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6(4):733–744, December 1977.

[BSV88] O. Berkman, B. Schieber, and U. Vishkin. Some doubly logarithmic optimal parallel algorithms based on finding all nearest smaller values. Technical Report UMIACS-TR-88-79, submitted to J.Algorithms, University of Maryland, 1988.

[BSVW86] S. Bublitz, U. Schürfeld, B. Voigt, and I. Wegener. Properties of complexity measures for PRAMs and WRAMs. *Theoretical Computer Science*, 48:53–73, 1986.

[BV89] O. Berkman and U. Vishkin. Recursive *-tree parallel data structure. In *FOCS*, pages 196–202, 1989.

[CDHR88] B. Chelbus, K. Diks, T. Hagerup, and T. Radzik. Efficient simulations between CRCW PRAMs. In *13th MFCS, Lecture Notes in Computer Science 324*, pages 230–239. Springer-Verlag, 1988.

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15:87–97, 1986.

[Cha91] S. Chaudhuri. Tight bounds for the chaining problem. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 62–70, 1991.

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

[CSV84] A. Chandra, L. Stockmeyer, and U. Vishkin. Constant depth reducibility. *SIAM Journal on Computing*, 13:423–439, 1984.

[DKR90] M. Dietzfelbinger, M. Kutylowski, and R. Reischuk. Exact time bounds for computing boolean functions on PRAMs without simultaneous writes. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 125–137, July 1990.

[DR86] P. Dymond and W.L. Ruzzo. Parallel RAMs with owned global memory and deterministic context-free language recognition. In *ICALP*, pages 95–104, July 1986.

[Eck79] D. Eckstein. Simultaneous memory access. Technical Report TR-79-6, Iowa State University, 1979.

[Edm91] J.A. Edmonds. Lower bounds with smaller domain size on concurrent write parallel machines. In *Structures in Complexity Theory*, pages 322–333, July 1991.

[EG88] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computing. *Ann. Rev. Comput. Sci.*, 3:233–283, 1988.

[FLRY89] F. Fich, M. Li, P. Ragde, and Y. Yesha. Lower bounds for parallel random access machines with read only memory. *Information and Computation*, 83(2):234–244, November 1989.

[FMW87] F. Fich, F. Meyer auf der Heide, and A. Wigderson. Lower bounds for parallel random access machines with unbounded shared memory. In F. Preparata, editor, *Advances in Computing Research*, volume 4, pages 1–15. JAI Press Inc., 1987.

[FR90] F. Fich and V. Ramachandran. Lower bounds for parallel computation on linked structures. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, July 1990.

[FRW88a] F. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM Journal on Computing*, 17:606–627, 1988. A preliminary version appeared in the Proceedings of the Third ACM Symposium on Principles of Distributed Computing, 1984, pages 179-189.

[FRW88b] F. Fich, P. Ragde, and A. Wigderson. Simulations among between concurrent-write PRAMs. *Algorithmica*, 3:43–51, 1988.

[FSS84] M. Furst, J.B. Saxe, and M. Sipser. Parity, circuits, and the polynomial-time hierarchy. *Math. Sys. Th.*, 17:13–27, 1984.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC*, pages 114–118, 1978.

[FW90] F. Fich and A. Wigderson. Towards understanding exclusive read. *SIAM Journal on Computing*, 19(4):718–727, August 1990.

[GNR89] E. Gafni, J. Naor, and P. Ragde. On separating the EREW and CROW models. *Theoretical Computer Science*, 68(3):343–346, 1989.

[Gol82] L. Goldschlager. A unified approach to models of synchronous parallel machines. *Journal of the ACM*, 29:1073–1086, 1982.

[Goo89] M. Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 127–137, June 1989.

[GR90] V. Grolmusz and P. Ragde. Incomparability in parallel computation. *Discrete Applied Mathematics*, 29(1):63–78, November 1990.

[GRS80] R.L. Graham, B.L. Rothschild, and J.H. Spencer. *Ramsey Theory*. John Wiley, New York, 1980.

[Has87] J. Hastad. *Computational Limitations for Small Depth Circuits*. MIT Press, Cambridge, Mass., 1987.

[HH86] J. Hartmanis and L. Hemachandra. Complexity classes without machines: On complete sets for UP. In *ICALP*, pages 123–135, July 1986.

[HKP] J. Hoover, M. Klawe, and N. Pippenger. Bounding Fan-Out in Logical Networks. *Journal of the ACM*, 31(1):13–18, 1984.

[HP89] S. Hornick and F. Preparata. Deterministic P-RAM simulation with constant redundancy. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 103–109, June 1989.

[HR90] T. Hagerup and T. Radzik. Every robust CRCW PRAM can efficiently simulate a PRIORITY PRAM. In *Proceedings of the Symposium on Parallel Algorithms and Architectures*, pages 117–124, 1990.

[KR90] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, volume A. MIT Press, 1990.

[KRS90] C. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, 1990.

[Kuc82] L. Kucera. Parallel computation and conflicts in memory access. *Information Processing Letters*, 14(2):93–96, 1982.

[LPV81] G. Lev, N. Pippenger, and L.G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Trans. Comput.*, C-30:93–100, 1981.

[LY86] M. Li and Y. Yesha. New lower bounds for parallel computation. In *STOC*, pages 177–187, 1986.

[LY87] M. Li and Y. Yesha. The probabilistic and deterministic parallel complexity of symmetric functions. In *ICALP*, pages 326–335, 1987.

[LY89] M. Li and Y. Yesha. New lower bounds for parallel computation. *Journal of the ACM*, 36(3):671–680, 1989.

[MR84] F. Meyer auf der Heide and R. Reischuk. Limits to speed up parallel machines by large hardware and unbounded communication. In *FOCS*, volume 25, pages 56–64, 1984.

[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.

[MW87] F. Meyer auf der Heide and A. Wigderson. The complexity of parallel sorting. *SIAM Journal on Computing*, 16(1):100–107, February 1987.

[Nis89] N. Nisan. CREW PRAMS and decision trees. In *STOC*, pages 327–335, 1989.

[NS81] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Trans. Comput.*, C-30:101–107, 1981.

[NS92] N. Nisan and M. Szegedy. On the degree of boolean functions as real polynomials. In *STOC*, 1992.

[PY91] I. Parberry and P. Yan. Improved upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 20:88–99, 1991.

[Raga] P. Ragde. unpublished manuscript.

[Ragb] P. Ragde. Processor-time tradeoffs in PRAM simulations. *Journal of Computer and System Sciences*, 44, 1992.

[Ran87] A. Ranade. How to emulate shared memory. In *FOCS*, pages 185–194, 1987.

[RSSW88] P. Ragde, W. Steiger, E. Szemeredi, and A. Wigderson. The parallel complexity of element distinctness is $\omega((\log n)^{1/2})$. *SIAM J. Dis. Math.*, 1:399–410, 1988.

[Rub] D. Rubinstein. personal communication.

[Ruz] W.L. Ruzzo. personal communication, cited in [BH89, Bea87].

[Sch80] J. T. Schwartz. Ultracomputers. *ACM Trans. Programming Lang. Systems*, 2:484–521, 1980.

[Sim82] H. Simon. A tight $\omega(\log\log n)$ bound on the time for parallel RAM's to compute nondegenerate boolean functions. *Information and Control*, 55:102–107, 1982.

[Smi90] Burton Smith. The tera computer system. In *Proceedings of the ACM International Conference on Supercomputing*, pages 1–7, 1990.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *STOC*, pages 77–82, 1987.

[Sni] M. Snir. private communication.

[Sni85] M. Snir. On parallel searching. *SIAM Journal on Computing*, 14(3):688–708, August 1985.

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

[SV84] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13:404–422, 1984.

[SV85] S. Skyum and L. Valiant. A complexity theory based on boolean algebra. *Journal of the ACM*, 32:484–502, 1985.

[SV90] B. Schieber and U. Vishkin. Finding all nearest neighbors for convex polygons in parallel: A new lower bound technique and a matching algorithm. *Discrete Applied Mathematics*, 29(1):97–112, November 1990.

[Sze89] G. Szegedy. *Algebraic Methods in Lower Bounds for Computational Models with Limited Communication.* PhD thesis, University of Chicago, 1989.

[Tar88] G. Tardos. Query complexity, or why is it difficult to separate $np^a \cap co - np^a$ from p^a by a random oracle a?, 1988. manuscript.

[TLR88] J. Trahan, M.C. Loui, and V. Ramachandran. Multiplication, division, and shift instruction in parallel random access machines. In *Proc. 22nd Conf. Inf. Sci. Syst.*, pages 126–130, 1988.

[Tra88] J. Trahan. *Instruction Sets for Parallel Random Access Machines.* PhD thesis, University of Illinois, Urbana-Champaign, 1988.

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

[Vis83] U. Vishkin. Implementation of simultaneous memory access in models that forbid it. *J. Alg.*, 4:45–50, 1983.

[VW85] U. Vishkin and A. Wigderson. Trade-offs between depth and width in parallel computation. *SIAM Journal on Computing*, 14:303–314, 1985. FOCS 1983, pages 146-153.

[WZ88] I. Wegener and L. Zádori. A note on the relations between critical and sensitive complexity. Technical Report 256, Universität Dortmund, 1988.