Chapter 7

Parallel Models and Implementations

models.tex

7.1 *Model?* What Model?

7.1.1 Components of a Parallel Computer

When referring to sequential algorithms, the question of "which model is used" seems foreign. The reason that we rarely mention a sequential model is that there is a more-or-less universally acceptable one that programmers use without question: the Random Access Memory model (Figure 7.1).

The RAM is a convenient simplification of modern uniprocessor systems. It contains a processing component (known as the Central Processing Unit (CPU) or simply the processor), a data-storing, addressable component (ucually called the Memory), and a connection between the two (known as

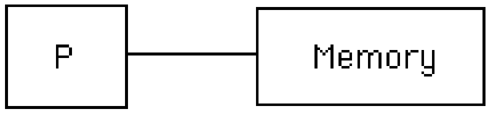

Figure 7.1: The RAM model of sequential computation contains the CPU, the Memory and a connection between them.

the *von-Neumann bottleneck*). The existence and the speed of the clock, the existence and the number of registers, the width of the bus, the size of the hard drive, etc, may be good reasons for someone to give up his 386-type PC to buy a Pentium-based machine, yet all these characteristics are almost never a factor when he is programming his computer. The RAM may be a gross oversimplification of how the Macintosh and the Cray-1 work, yet it is a sufficient description from the programmer's point of view, because it points out all the important aspects of the sequential machines and none of the non-important: That the location where the data are stored and the location where they are processed are distinct; and that they are moved between these two locations one-at-a-time (or at least a few-at-a-time).

The question that naturally arizes is, can we do something similar for the existing parallel machines? So far, we have only said that a parallel computer contains many processors. However, when you throw thousands of processors into the picture, things are not as simple. There are many questions that suddenly arize:

1. Does the machine have one (big) Memory, or many (small) ones? Or does it have both?

2. Are some (or all) of the memory locations shared by all the processors or not?

3. Are some memory locations closer to particular processors or are all equidistant from them?

4. Do the processors use memory locations to communicate, or do they send messages to each-other using some interconnection network?

5. Do the processors work in a synchronous manner or do they operate completely independently from each other?

6. Are all of the processors of the same type, or are they different?

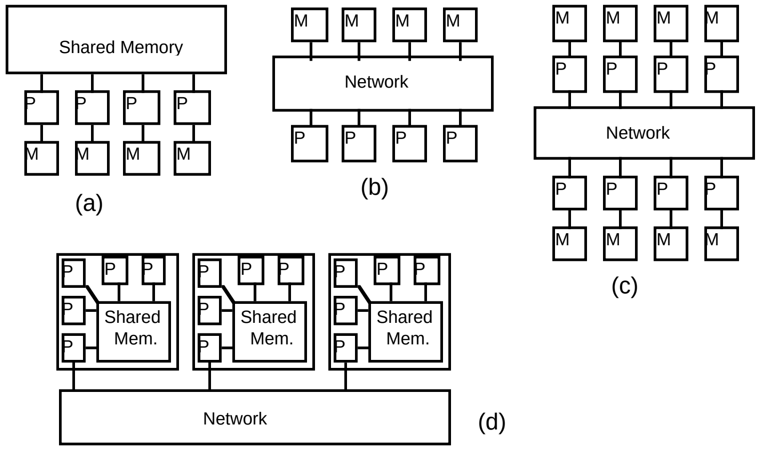

Figure 7.2: Four of the many possible ways to connect processors into a parallel computer: (a) RAMs have been hooked to a large shared memory; (b) CPUs communicate with shared memories through some wires composing an interconnection network; (c) RAMs are connected through a network, no memory is shared here; (d) Chips containing a few processors with their own memory are communicating using a network. A particular processor is used as gateway to the network.

There are many questions of that sort that one can ask, and many answers that one can give. See figure 7.2 for some potential answers. In fact, the manufacturers of parallel machines rarely give identical answers to these questions. Very often, the only thing that their designs have in common is that they contain more than one processors.

As a result of that trend, learning how to program a particular parallel computer says nothing about your ability to program another. This is not a pleasant situation, but there are things we can do to amegliorate the problem. For example one can learn the *principles* of designing parallel algorithms, what is usually called “learning to think in parallel.” Then, one can apply these principles to utilize a particular parallel computer. This book takes such an approach.

In the next sections we will consider a simple and elegant model of parallel computation which will help us learn those principles, because it hides many irrelevant details of any particular machine. This is the Parallel-RAM (PRAM) model which, like the sequential RAM, oversimplifies the components of a parallel computer. We will use this model to design several basic parallel algorithms using the PRAM-L language. We will, also, describe the architecture of the MasPar computer, a particular parallel machine with an architecture that little resembles PRAM. However, we will see how the algorithms developed on the simple PRAM model can be modified to be programmed on the MasPar using the MPL language.

7.1.2 Flynn's Classification Scheme

Every time a new parallel computer appears, the first question in the `comp.parallel` network group is "Is it a SIMD or a MIMD?" In this section will examine these terms that, even though are not particularly important, have obsessed many.

One of the earlier attempts to classify parallel computing system is due to Flynn. His classification scheme is described in terms of two independent factors: the number of *data streams* that can be simultaneously processed, and the number of *instruction streams* that can be simultaneously processed. By "data stream" we mean the different pieces of data that are being operated upon at a given moment by the processors, while by "instruction stream" we mean the different instructions that are being executed simultaneously by the processors. According to Flynn's classification, every computer falls into one of four categories: SISD, SIMD, MIMD and MISD:

SISD. As you know, uniprocessor machines can execute a single instruction affecting a single piece of data at any time, therefore they are called single instruction stream, single data stream (SISD) machines.

SIMD. Some machines are composed of a large number of identical processors that execute *synchronously* the same instruction; each one,

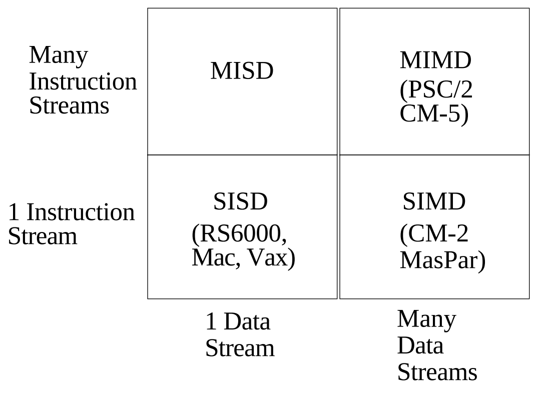

Figure 7.3: Flynn's classification.

however, operates on a different piece of data. The single instruction stream, multiple data streams (SIMD – pronounced "seem-dee") group contain the so-called vector processor (or array processor) machines, like the MasPar.

MIMD. This group contains all the computer systems containing processors that operate *asynchronously*, i.e., each processor may execute a different instruction while using a different piece of data. Therefore, the MIMD (pronounced "meem-dee") group covers the range of multiprocessor systems, and is a rather coarse classification of many different machines. As a matter of fact, there have been some further refinements on this class that take into account the memory system and communications methods, but we will not examine them here.

MISD. This chracterization refers to machines that have not been built and may never be, and exists only for the complete description of the classification system. Even though some argue that a cluster of workstations executing different programs on the same file (one that contains, say, DNA data or large numbers to be factored) behave as MISD, they hardly qualify as *a single* machine.

Even though Flynn's classification defines four names, only two of them are relevant to parallel computers. That's why the question is whether a machine is a SIMD or a MIMD, ignoring the other two categories. Of them, the SIMD systems are considered easier to program, because the programmer has to think of only a single thread of execution. On the other hand, the MIMD machines are considered more efficient because you can take advantage of the full machine power. Newer systems, like the CM-5, try to combine the benefits of both worlds by using the SIMD idea in programming while the machines actually executes in MIMD mode.

Stack "Real Processors"

7.2 Model? *Which* Model?

7.2.1 Characteristics of Programming Models

It is now time to define a convenient programming model that will help us think in abstract terms when we design parallel algorithms. First, we have to identify what are the major machine characteristics that a programmer needs to know in order to program in it. As you may have observed, the major issue in Flynn's classification is the machine's **operating mode**, i.e., the existence or lack of synchronicity in the hardware components. Apparently, the operating mode affects the way machines are programmed: If all the processors operate synchronously, then they run the same program. Therefore we only need to think about the program that a single processor executes, and have every processor execute it line-by-line. If, however, the machine operates asynchronously, then we may have to write many programs, one per processor (or group of processors) and to define explicitly the points in the code where the processors need to synchronize their work.

In order to cooperate in the solution of a problem, the processors need to communicate, and the programmer has to know how this happens. So, the processors' **communication mechanism** also affects the way machines are programmed. If the processors use messages to exchange data and status information, we have a *message passing* model. Alternatively, processors may communicate by reading and writing messages to some *memory space*: the processor that wishes to send a message, writes the message into some memory location from which the other processor can read it.

Finally, all programs use variables which they keep in memory, and the programmer may need to know where these variables are kept in order to access them. We are referring to the **logical view of the memory space** — not to the physical way in which the memory is handled by the hardware. If the memory space is viewed as a contiguous, uniformly addressed entity (much like the memories of the sequential processors), the model is called a *shared memory* model. In this case, the programmer can write, for example, $A[2]$, and there is no ambiguity about this memory location. Otherwise, if the memory is viewed as being distributed among the processors, the programmer may need to also mention the address of the processor that contains $A[2]$. In this case, the model is called a *distributed memory* model. You should note that we make a clear distinction between the logical view of the memory space and the physical/actual way in which the memory is handled by the hardware, because the latter is not relevant to the programmer.

In message passing and distributed memory models, the programmer may need to know the topology of the *interconnection network* that connects the processors and the memories, in order to send the messages or the variable requests. Mesh, Hypercube, cube-connected-cycles, shuffle-exchange and butterfly are examples of these networks, and we will examine them in Part III.

So, we need to specify at least three things to have a clear idea of the programming model we are using.

1. Operating mode: Synchronous versus Asynchronous

2. Communication method: Message passing versus memory

3. Logical view of memory: Shared versus distributed.

There are many programming models that we can define, and you may wonder what is the simplest of all these models — where by “simple” we mean the “easier to think about”. A synchronous model is simpler than an asynchronous one; a model that uses the memory space for communication is simpler than one that uses messages; and a shared memory model is simpler than a distributed memory one. The model that has all these three characteristics is the PRAM, which we will describe next.

About *data-parallel* programming here?

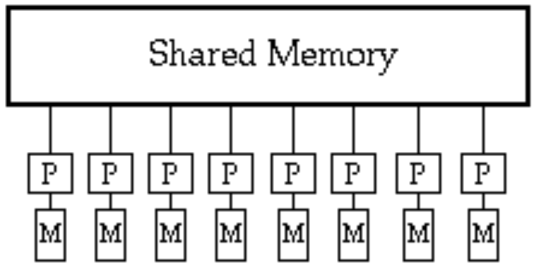

Figure 7.4: The PRAM model.

7.2.2 The PRAM model

The PRAM is an idealized parallel machine which was developed as a straightforward generalization of the sequential RAM. Because we will be using it often, we give here a detailed description of it.

Description

A PRAM uses p identical processors, each one with a distinct id-number, and able to perform the usual computation of a sequential RAM that is equipped with a finite amount of local memory. The processors communicate through some shared global memory (Figure 7.4) to which all are connected. The shared memory contains a finite number of memory cells. There is a global clock that sets the pace of the machine executon. In one *time-unit* period each processor can perform, if so wishes, any or all of the following three steps:

1. **Read** from a memory location, global or local;
2. **Execute** a single RAM operation, and
3. **Write** to a memory location, global or local.

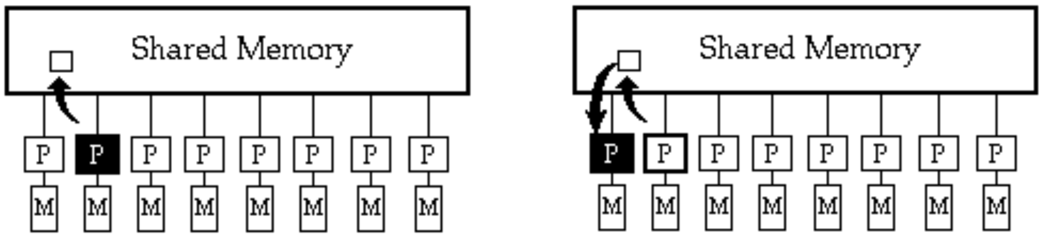

Figure 7.5: Communication between two processors in the PRAM model takes two time-units. Here, the two leftmost processors communicate. In the first step one processor writes a message to some memory location (left). In the second step, the other processor reads the message. It is the job of the programmer to specify the memory location used for the communication.

If processor p_i wants to communicate with processor p_j, it can do so by writing to some memory location from which p_j will read. (See figure 7.5.) So, the model makes the assumption that communication between processors takes constant time (two time-units).

An example: Adding Matrices

Let us see how the PRAM works, with a simple example: Adding two matrices A and B to produce a new matrix SUM using a PRAM with p processors. We assume that A and B have been loaded into the shared memory, and each matrix contains n integers. To make things simple, we also assume that $n \leq p$. How does the PRAM performs the addition?

We allocate n consecutive memory locations indexed 1 through n into the shared memory to represent the array SUM. We assign one processor per memory location, so that processor with id-number i is assigned to location $SUM[i]$. We can do that since we have enough processors ($p \geq n$). The

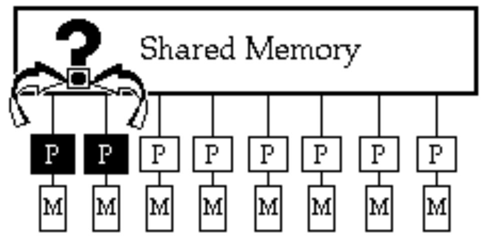

Figure 7.6: A conflict occurs when two or more processors try to access the same memory location for reading or writing.

remaining processors, if any, will remain idle throughout the computation.

Now, each non-idle processor with id-number i will execute the following three steps:

1. Read $A[i]$ into some register;
2. Read $B[i]$ and add it to the register;
3. Write the contents of the register into $SUM[i]$.

Note that the parallel algorithm we described takes only 3 steps, while a sequential algorithm would take $3n$ steps. Of course, it also uses n processors while the sequential would only use one.

Handling Concurrent Accesses

The fact that the communication between the processors in the PRAM is implemented via accesses to the shared memory makes access conflicts possible. These conflicts may be caused by concurrent accesses to a memory cell by two or more processors for either reading or writing. (See figure 7.6.)

Depending on the way the access of the processors to the global memory is handled, PRAMs are classified as EREW, CREW and CRCW.

EREW. In the exclusive-read-exclusive-write (EREW) PRAM model, no conflicts are permitted for either reading or writing. If, during the execution of a program on this model, some conflict occurs, the program's behavior is undefined.

CREW. In the concurrent-read-exclusive-write (CREW) PRAM model, simultaneous readings by many processors from some memory cell are permitted. However, if a writing conflict occurs, the behavior of the program is undefined. The idea behind this model is that it may be cheap to implement some broadcasting primitive on a real machine, so one needs to examine the usefulness of such a decision.

CRCW. Finally, the concurrent-read-concurrent-write (CRCW) PRAM, the strongest of these models, permits simultaneous accesses for both reading and writing. In the case of multiple processors trying to write to some memory cell, one must define which of the processors eventually does the writing. There are several answers that researchers have given to this question. The most commonly used are:[1]

COMMON. All processors attempting to write to a memory location must write the same value, so the outcome of some writing is the same, no matter which processor wins. This is called the COMMON CRCW PRAM model.

ARBITRARY. Any processor may win. No restriction is placed on the values written. This is called the ARBITRARY CRCW

[1]Two more CRCW PRAM models appear sometimes in the literature: The WEAK and the STRONG CREW PRAM. In the WEAK, all processors trying to write to a memory location must write the value zero; no other value is permitted. In the STRONG model, the value that is written in the memory cell is the largest (or equivalently the smallest) of all the values attempting to be written.

PRAM. Note that, if the program runs a second time, the outcome of a concurrent writing may be different. However, the outcome of the computation in *every* case must be correct.

PRIORITY. In this model we assume that each processor has some unique value *id* associated with it, the *id*'s being drawn from a totally ordered set. In case of a concurrent write, the processor with the lowest (or, equivalently, the highest) *id* wins. This is called the PRIORITY CRCW PRAM model.

7.2.3 Relations Between PRAM Models.

Given the plethora of PRAM models, the need for comparison and simulation among the models arises. In fact, there is a hierarchy in the PRAM models. In particular,

Theorem 6 *The EREW PRAM model is less powerful than the CREW model, which in turn is less powerful than the CRCW model. The hierarchy is extended inside the CRCW model where the COMMON is less powerful that the ARBITRARY, which in turn is less powerful than the PRIORITY CRCW PRAM model.* [2] *(Figure 7.7).*

Proof. This theorem is easy to prove from the definitions of the models: Every time we defined a model that comes in the hierarchy after model A, we threw in some new features. (For example, we defined the CREW after the EREW, to which we allowed concurrent reading, something that was not allowed in the EREW.) So, a program written for some model A can run on models stronger than A, without any change: The program will just not use any of the extra features that the stronger model provides. □

What we proved is that, any model that comes later than A in the hierarchy is at least as strong as A. It turns out that the hierarchy is *strict*:

[2] In this hierarchy, Weak is the weakest of the CRCW models and Strong is the strongest of them.

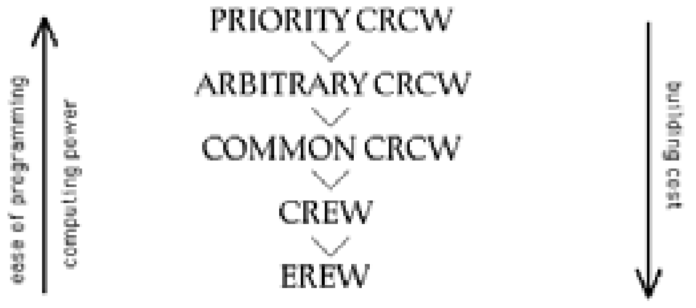

Figure 7.7: The basic PRAM Hierarchy.

a model B that comes after A is *strictly more powerful* than A. In other words, there are problems that can be solved in the B model with algorithms that have better complexity than any algorithm for these problems in model A. To see an example of such a problem we will need the following non-trivial fact, which we mention without proof. For a proof, consult [**?**].

Fact 1 *Computing the OR of n bits requires $\Omega(\lg n)$ time on a CREW PRAM, no matter how polynomially many processors are available or how much memory is used.*

Observe that the OR function is computed in $O(1)$ time on a COMMON CRCW PRAM, while it requires $O(\log n)$ time on a CREW PRAM. Therefore we have the following theorem:

Theorem 7 *Concurrent write is strictly more powerful than exclusive write.*

□

You may wonder what this hierarchy is good for, apart from its theoretical interest. As we mentioned, if model B comes later than model A in the hierarchy, it contains more "features" than A. Implementing these feartures increases the cost of building the machine. For example, you may think of a CREW PRAM machine, as a EREW PRAM with some extra circuitry that allows concurrent reading. This extra circuitry could be some global broadcasting mechanism that sends a value simultaneously to every processor. You may also think of a CRCW PRAM machine as a CREW PRAM which in addition provides some circuitry for resolving concurrent writing conflicts, that is considered more expensive to build than the global broadcasting mechanism. And so on. The cost of building such mechanisms changes with the advances of technology but, in general, researchers believe that *the stronger the machine, the more expensive it is to build it.*

So, to argue about whether it is worth to build a stronger machine or not, one has to know what *will be gained* by the stronger machine. To address this question, we will break it down into two other questions:

1. EREW vs CRCW: How much more powerful the strongest model is when compared to the weakest model?

2. COMMON vs PRIORITY: Among the CRCW models, how much more powerful the strongest model is when compared to the weakest model?

EREW vs CRCW

We will prove the following theorem:

Theorem 8 *Any algorithm written for the PRIORITY CRCW model can be simulated on the EREW model with a slowdown of $O(\lg r)$ where r is the number of processors employed by the algorithm.*

Proof. We have an algorithm that runs correctly on a PRIORITY CRCW PRAM, and a EREW PRAM machine on which we want to simulate the algorithm. If we were to execute the algorithm without modification on the EREW machine, it would not work. The problem would not be in the

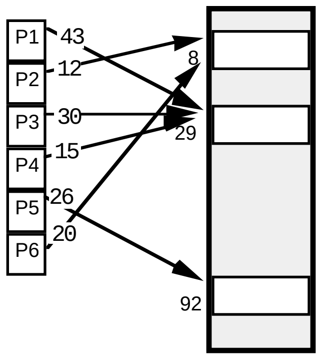

Figure 7.8: Example of write conflicts caused by code written for the PRIORITY CRCW PRAM model. Six processors try to write into three memory locations, as the arrows show. The labels on the arrows indicate the values that they try to write. There are two conflicts, at memory locations 8 and 29.

executable part of the code, since both machines understand the same set of executable statements. Instead, the problem would be in the statements that access the shared memory for reading or writing. For example, every time the algorithm says:

Processor P_i reads from memory location y into x

it might involve a concurrent reading which the EREW machine cannot handle. The same is true for a concurrent write statement (depicted visually in figure 7.8). In order to fix the problem, we have to simulate each statement of that sort into a sequence of statements that do not involve any concurrency, but have the same result as if we had concurrency.

Let us assume that the algorithm uses r processors, named $P_1, P_2 \ldots, P_r$; the EREW machine also uses r processors. The EREW machine, however, will need a little more memory: r auxiliary memory locations $A[1..r]$, which will be used to resolve the conflicts.

The idea is to replace each fragment code of the algorithm:

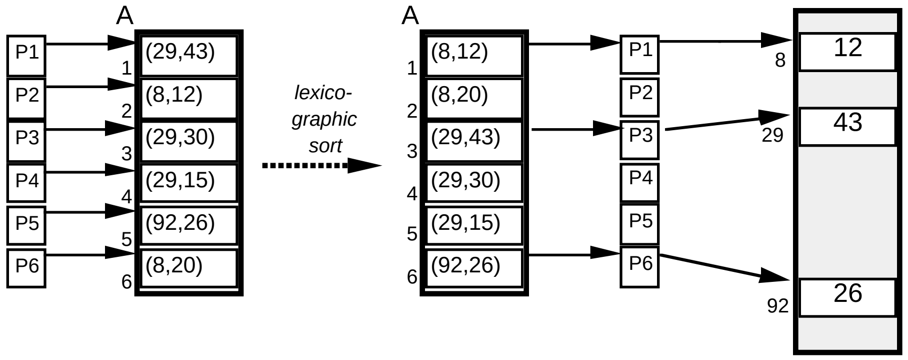

Figure 7.9: Simulation of a PRIORITY CRCW on a EREW PRAM. On the left, processors write their requests on auxiliary memory. These entries are sorted and then the processors see the result of the sorting. On the right hand side, the processors that are permitted, do the write.

Processor P_i accesses (reads into x or writes x into) memory location y

with code which:

a) has the processors request permission to access a particular memory location,

b) finds out, for every memory location, whether there is conflict, and

c) decides which processsor of the competing will do the access.

This is achieved by the following fragment of code:

```
1.    Processor $P_i$ writes $(y, i)$ into $A[i]$
2.    Auxiliary array $A[1..r]$ is sorted lexicographically in increasing order.
3.    Processor $P_i$ reads $A[i-1]$ and $A[i]$ and determines whether it is
                 the highest priority processor accessing some memory location
4.    If $P_i$ is the highest priority processor, then:
          If the operation was a write, $P_i$ does the writing
```

Else P_i does the reading, and the value read is propagated
    to the processors interested in this value.

The last step takes $O(\lg r)$ time. The sorting step also takes $O(\lg r)$, as the following non-trivial fact shows. (We mention it here without proof. For a proof, consult [**?**].)

Fact 2 *Sorting n numbers on a EREW PRAM machine using n processors requires $O(\lg n)$ parallel time.*

□

COMMON vs. PRIORITY

We will prove the following theorem:

Theorem 9 *Each algorithm for the r-processor PRIORITY CRCW model can be simulated by a COMMON CRCW machine with no loss of parallel time, provided that $O(r^2)$ processors are available.*

Proof. We have an algorithm that runs correctly on a PRIORITY CRCW PRAM using r processors, and we want to execute it on a COMMON CRCW PRAM machine. As with the previous simulation, the problem is at the memory access statements:

Processor P_i writes x into memory location y

which the algorithm may contain.

The COMMON CRCW machine cannot handle a situation in which two processors P_i and P_j (or more) write into memory location y different values.

Assume $i < j$. To resolve the conflict correctly, we need to simulate the write statement so that P_j will not attempt to write. Note that P_i will do

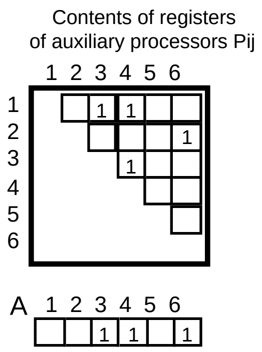

Figure 7.10: Simulation of a PRIORITY CRCW writing step on a COMMON CRCW PRAM (part 1). The auxiliary processors P_{ij} detect the conflicts, and they indicate that, by writing 1 into the array A (shown below).

the write, unless there is another processor P_h, with $h < i$, which also wants to write into y.

To achieve that, we need to do $O(r^2)$ comparisons, one for every potential conflict of each pair of processors. We can do that in constant time by employing $\frac{r^2}{2} - r$ auxiliary processors, which we will call P_{ij}, for $1 \leq i < j \leq r$ (See figure 7.10). We will also use some auxiliary memory $A[1..r]$ which is initialized to 0 in constant time.

The simulation proceeds as follows (shown in figures 7.10 and 7.11):

- Code for Processor P_{ij}, $1 \leq i < j \leq r$:

1. Reads into a register the location x_i that P_i wants to write.

2. Reads the location x_j that P_j wants to write, and compares it with the contents of the register.

3. If $x_i = x_j$ (there is a conflict), writes 1 into $A[j]$ (meaning that P_j lost the conflict and should not attempt to write). Note that this is a concurrent write statement which the COMMON machine can handle.

- Code for Processor P_i, $1 \leq i \leq r$:

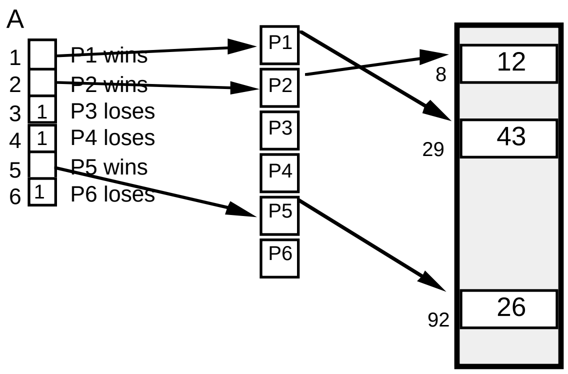

Figure 7.11: Simulation of a PRIORITY CRCW writing step on a COMMON CRCW PRAM (part 2). Array A is shown again, with comments about the meaning of its values. Processors read the A entries and do the writing without conflicts.

1. Tests whether $A[i]$ is still 0; if it is, P_i is permitted to do the concurrent write that the algorithm calls for.
2. Else it las lost at least one conflict and does not attempt to write.

□

So, we can do the simulation at no extra time, but we need to use $O(r^2)$ more processors. Alternatively, we could do the simulation with no more processors, but paying an extra $O(\log r)$ factor in running time.

7.2.4 The PRAM-L language

To be able to convert the PRAM algorithms into algorithms for existing parallel machines, we will need to be a little more detailed about them. For this reason we will describe here the syntax of PRAM-L, a Pascal-like pseudocode for the PRAM model. There is no compiler for PRAM-L currently.

You should keep in mind the following points:

1. For simplicity we will not show variable declarations in the PRAM-L.
2. The expression `<List of statements>` denotes one or more statements that should be executed in a block.

As you may have noticed, the pseudocode we have given so far (in section 7.2.3) seems a little awckward because of the processors' scheduling details: we described a different program for each processor, instead of a single program that every processor had to execute. So, we were writing about the actions of processor P_i, $1 \leq i \leq r$ and processor P_{ij}, $1 \leq i < j \leq r$.

Giving the scheduling details for each processor becomes quickly cumbersome. PRAM-L takes another approach: When the scheduling details are not difficult to figure out we omit them. It is assumed that in the beginning of each statement, each processor is assigned some memory location, and executes the statements that assign value to that memory location. However, when the scheduling details are not trivial, we describe them.

The syntax of the execution statements is described by the following components:

Selecting Active Processor Set statement.

for all $i \in V$ **in parallel do** `<List of statements>`

The condition $i \in V$ is evaluated by all the processors, and those evaluating "true" will execute the `<List of statements>` that follow.

Processor evaluating "false" will remain idle during the execution of the `<List of statements>`.

Assignment statement.

let varname $\leftarrow$ expression

Compute the expression in the right-hand-side, and assign the result to the variable of the left-hand side. The variable could be in local memory or in shared memory.

Conditional statement.

```
if condition then
  <List of statements - 1>
  { else <List of statements - 2> }
end if
```

Every active processor evaluates the condition. Those evaluating "true" will execute the `<List of statements - 1>`. Those evaluating "false" will execute the `<List of statements - 2>` (if there is one) after the `<List of statements - 1>` has finished. Note that the semantics of this statement are different than the semantics of the sequential programs, since you may have both statements executed!

Infinite iteration statement.

```
while condition { in parallel } do
      <List of statements>
end while
```

Every active processor evaluates the condition. While there is still some active processor evaluating "true", the `<List of statements>`

is executed. When no active processor exists, i.e., when all processors evaluate “false” the condition, the loop stops.

Definite iteration statement.

```
for var-name $\leftarrow$ initial-value to final-value { in parallel } do
   <List of statements>
end for
```

The loop will be executed for the number of times specified, even if nothing happens, i.e., if there are no active processors to execute the `<List of statements>`.

Procedure Declaration statement.

Procedure proc-name (formal-parameter-list) statement

It has the usual semantics that sequential languages have.

Procedure Call statement.

call proc-name (actual-parameter-list)

The parameter list is first evaluated, and then control is turned to the procedure called.

Now we can see how we could express some of the code of section 7.2.3 into PRAM-L. The code

Processor P_i reads from memory location y into x

of the CRCW PRAM algorithm, simply becomes

```
for all $i$, $1 \leq i \leq r$ in parallel do
    $y_i \leftarrow x_i$
```

while the simulation is

```
**for all** $i$, $1 \leq i \leq r$ **in parallel do**
    $A[i].1 \leftarrow y_i$
    $A[i].2 \leftarrow x_i)$
    sort(A)
    **if** $(A[i-1].1 \neq A[i].1)$ **then**
        $y_i \leftarrow x_i$
```

In the notation above, we store a record inside each memory location $A[i]$ and we access its two members with the usual “dot” notation.

As you may have observed, it is not obvious from the code whether some statement involves concurrent memory access or not; determining it is part of careful algorithm analysis.

Notes. Some authors define the operations *readSM* and *writeSM* to explicitly read from the shared memory and write to it. This is very useful if one wants to count the shared memory accesses for evaluation purposes, but makes the code unnecessarily involved, so we do not take this approach.

Also, there is no function call defined in PRAM-L, even though programming languages rely heavily on them. Again, the reason we omit them is for keeping the syntax simple.

7.3 A Parallel Computer

7.3.1 The MasPar Architecture

Description of the MasPar Architecture and network goes here.

7.3.2 The MPL Programming Language

Description of the MPL goes here, along with examples of pixel-averaging, summing up values, and timing.

7.3.3 How to translate PRAM-L into MPL

General rules, not every possibility is covered.

7.4 Measuring Parallel Performance

7.4.1 Time, Space, Work

As with sequential programs, we need some method of measuring the performance of parallel programs. Recall that for sequential programs, we have two metrics to measure their performance:

Sequential Time $t_s(n)$. The time period elapsed between the beginning and the end of the algorithm execution, in the worst case. Often it is calculated theoretically by the number of basic steps (operations) required for the execution of the algorithm, and is measured in asymptotic terms (i.e., using the big-O notation).

Sequential Space. The size of the Main memory that is required for the execution of the algorithm.

For example, sorting a file with n keys using mergesort takes time $O(n \log n)$, and requires $n + 2$ memory cells. Over the years, these metrics have been proven useful and have not changed (even though Space does not play as an important role today because Main memory have become much cheaper).

Time and Space, defined appropriately, also play an important role in evaluating parallel programs. However, to account for the fact that we are using multi-processor computers, we have to also take into account the number of processors used:

Parallel Time $t_p(n)$. The time period elapsed between the beginning of the first processor and the end of the last processor during the execution of the algorithm, in the worst case. The smaller the time, the better the algorithm. As with sequential time, parallel time is a function of the input size n, since the larger the problem, the longer it takes to be solved. Parallel time is expressed theoretically by the number of basic steps composing the *longest path* of calculations for the execution of the algorithm. The size of the longest path depends, often, on the number of processors used.

Number of Processors $p(n)$**.** The number of processors required by the algorithm. As with time, it is often a function of the input size n, since more input can be processed at once, if more processors are available.

Space. The sum of the sizes of all the Main memories involved (shared or not) in the execution of the algorithm.

Since the time taken by a parallel algorithm is a function of the number of processors employed, we have to examine them in connection. For this reason we define the notion of *work*, described next.

Work

Parallel time is, in general, affected by the number of processors employed. When more processors are available, the algorithm may be able to make use of them and run faster. When fewer processors are available, it may take longer to finish the calculations, since some processors may need to do the work that others would otherwise. To express this interconnection between time and number of processors, we have the notion of Work:

We say that a parallel algorithm for some problem performs *work* $w(n)$, if it runs in parallel time $t_p(n)$ using $p(n)$ processors, and

$$w(n) = t_p(n) \times p(n)$$

Work is one of the fundamental metrics in parallel computing, and it is used for charging computer usage at national supercomputing centers. Observe, however, that $w(n)$ represents an *upper bound* on the total number of steps executed by all the processors employed. We call it an "upper bound," since the definition assumes that every processor is active in every step. This may not be the case, as some processors may not do any useful work during some steps, but just wait for others to finish. We will see such cases later.

Work versus Sequential Time. Theoretically, you can always simulate the behavior of a parallel algorithm that uses p processors, $P_1, P_2, \ldots, P_p$ on a sequential machine, using the *sequential simulation technique*: The processor of the sequential machine executes the first step that P_1 would execute, then the first step that P_2 would execute, etc., until the first step that P_p would execute. Then, it continues with the second step of each processor, and then the third, etc., until the last step of P_p.

Now we can see the relation of the work performed by a parallel algorithm, to the notion of sequential time: Consider a problem whose fastest known sequential algorithm has running time $T(n)$ for an input of size n. The definition of work, along with the sequential simulation of a parallel algorithm implies

$$w(n) = \Omega(T(n)).$$

If this were not true, then one could design a sequential algorithm with running time less than $T(n)$ by simulating the steps executed by the parallel algorithm! This contradicts the assumption that $T(n)$ was the fastest known sequential algorithm.

Work-efficiency and work-optimality. To express how close the parallel work can come to sequential time, we have the notions of work-optimal and work-efficient algorithms. We will use the term *polylogarithmic* of a quantity to express some power of the logarithm of that quantity. For example, $\log(n)$, $\log^{1/3}(n)$, $\log^3(n)$ are all polylogarithmic expressions on n.

We say that a parallel algorithm is *work-optimal* iff

- its parallel running time $t(n)$ is polylogarithmic on n, and
- its work is $w(n) = O(T(n))$.

The second condition, along with the above observation, implies that for work-optimal parallel algorithms, $w(n) = \Theta(T(n))$. Moreover, the sequential simulation technique implies that it is possible to create new sequential algorithms from work-optimal parallel ones.

Example or exercise: new sequential algorithm for calculating the sum of n numbers.

Example 5 *According to fact 1, sorting is work-optimal, since sequential sorting requires* $\Theta(n \log(n))$ *time.*

Work-optimality is highly desirable to designers of parallel algorithms. When it is not possible to come up with work-optimal algorithms, one can settle for work-efficient algorithms, that perform work which comes within a polylogarithmic fraction of work-optimality. So, a parallel algorithm is *work-efficient* iff

its running time $t(n)$ is polylogarithmic on n, and

its work $w(n)$ is within a polylogarithmic factor of $T(n)$.

Example 6 *Computing the connected components of a graph with* n *vertices and* m *edges on a CRCW PRAM in parallel requires* $O(\log n)$ *time and* $n+m$ *processors. Thus, this is a work-efficient algorithm.*

7.4.2 Other Metrics

There are three more metrics that are often used because they say important things about the parallel programs: Speedup (i.e., how much faster the algorithm is running when compared to a sequential algorithm), Efficiency (i.e., how well the algorithm uses the processors), and Scalability (i.e., how the algorithm behaves when more processors are available). We will examine each one of them in detail.

Speedup

In general, *speedup* S compares the parallel running time t_p of an algorithm that uses p processors to solve a particular problem Π, to the sequential running time t_s of an algorithm for the same problem:

$$S = \frac{t_s}{t_p}$$

The above definition is rather ambiguous: Does t_s refer to *any* sequential algorithm for the problem Π, the *best* algorithm for Π (if it exists), or the parallel algorithm simulated on a single processor? There are several options here. The following three types of speedup have been extensively used in the literature:

Relative Speedup. $t_s = t_1$ is the running time of the parallel program when it runs on a single processor of the parallel machine. It is not a very good measure of the performance of the algorithm, since the program contains code that does processor management, which is not needed when you only have one processor.

Real Speedup. $t_s = t^*$ is the running time of the *best* algorithm for problem Π on some sequential machine. This is a good but tough challenge for the parallel program, since it compares its performance to the performance achieved by the best sequential solution available.

Absolute Speedup. $t_s = t_1^*$ is the running time of the best sequential program when it runs one a single processor of the very same parallel machine. (Notice that the difference between absolute and real speedup is the machine used for measurement.) This is also considered a good measure of the performance of a parallel program.

Which of the three definitions is preferable? Clearly, Relative Speedup is not a very good measure, so we will not use it in this book. Real Speedup is a good measure because it shows how good a parallel solution for Π is compared to the best sequential solution on the best sequential machine. It is not, however, very well defined because sometimes one does not know what the best algorithm for Π is, and because it changes everytime you change the sequential machine. To define it better, for the programs that we will be running on the MasPar, we will use the ACU for running the sequential programs. Absolute Speedup is a fairer metric, since it factors out the performance of the best sequential machine and focuses on the performance of the parallel algorithm itself. We will use it in our measurements as well.

The previous three definitions refer to actual implementations of algorithms. When analysing the theoretical performance of an algorithm, however, the following definition is often used:

Asymptotic Real Speedup. $t_s = t^*(n)$ is the theoretical time of the best sequential algorithm when the input has size n, and $t_p = t_p(n)$ is the theoretical time of the parallel algorithm on the same input.

Apparently, the value of Real Speedup (and Asymptotic Real Speedup) should be between 1 (when the parallel algorithm does not run any faster than the sequential one) and the number of processors p (when all p processors are utilized optimally). So we can write

$$1 \leq S_p \leq p.$$

Logically, Real Speedup cannot exceed p, because of the sequential simulation technique. There are however, some *anomalous* cases in which the speedup exceeds p. These happen because of the memory hierarchy: Cache memory is faster than main memory, which in turn is faster than the disk. Parallel machines have more main memory and more cache than sequential machines, and can run larger instances of a problem without having to fetch data from the disk, while the sequential machine may have to go to the disk very often. Sequential simulation would not performe as well as it was expected, due to the memory hierarchy, but also because cycling to simulate the work of p processors will cause lots of expensive context switches.

Efficiency

Sppedup tells us how well the parallel algorithm performs compared to the sequential algorithm, but does not take into account the number of processors that the parallel algorithm is using. To get a feeling of how well the processors are utilized, we divide the Speedup with the number of processors employed by the parallel algorithm to arrive at the the quantity

$$E_p = \frac{S_p}{p}$$

which we will call *Efficiency*. As you may observe, $E_p \leq 1$. When the efficiency of an algorithm is close to 1, it means that each of the processors is doing useful work, and so it is highly desirable. Since the definition is based on Speedup, we have the analogous definitions: Relative, Real and Absolute Efficiency.[3]

An alternative definition for efficiency, is the ratio of the sequential time to solve a problem to the work of the parallel algorithm. The two definitions are equivalent, as we can see by applying the definition of Speedup on the previous definition: $E_p = \frac{S_p}{p} = \frac{t_s}{t_p \cdot p}$. Finally, Asymptotic Real Efficiency is defined as $E_p(n) = \frac{t^*(n)}{p(n) \cdot t_p(n)}$.

[3]Authors do not agree on the definition of efficiency and speedup; so one has to be careful when compares results.

Scalability

In addition to speedup and efficiency, it is useful to know whether the algorithm imposes any limit on parallelism, i.e., whether or not the algorithm continues to work as well larger problem instances when more processors are available. For example, assume we were able to get some speedup when we run an algorithm using 1000 processors. Now we have a problem instance twice as large as the previous one, but we can also afford twice as many processors. Can we still get a speedup? If this is the case, we say that the algorithm *scales up*, or is *scalable*. Otherwise, if the rate at which the algorithm operates eventually reaches a limit, we say that the algorithm is not scalable.

Formally, an algorithm is *scalable* if its parallel execution time is asymptotically smaller that the best serial time (i.e., $t_p(n) = o(t_s(n))$), independelntly of the number of processors used. To see when this condition may fail, i.e., when an algorithm may not scalable, consider a parallel algorithm that is composed of a number of phases, and assume that one of them cannot be parallelised. No matter how much speedup may be achieved by the remaining phases, and how little time the non-parallelizable phase may take, there will be some large instance of a problem that this non-parallelizable phase will dominate the computing time, no matter how many processors are available.

Scalability is a desirable property, because it promises "longevity" to the parallel algorithm: As machines become larger, non-scalable algorithms will become useless, no matter how good speedup they can achieve today.

7.4.3 *Amdahl's law

The section on scalability indicated that the existense of sequential phases within a parallel program may restrict the speedup we can achieve considerably. This idea was formulated into what is known as Amdahl's law, described below.

Any program that will run on parallel machines will have a sequential part and a parallel part. How the size of each one of these parts affects the speedup we get? Let us assume that R_p is the *rate* (in flops) at which the parallel part is executing, and R_s is the rate at which the sequential part is executing. Furthermore, let us denote with f the fraction of total execution time that the parallel part occcupies over the whole program. Then, the sequential part will be described by the fraction $1 - f$.

The rate $R(f)$ that the whole program will execute, given this particular f, is described by the following relation:

$$R(f) = \frac{1}{\frac{f}{R_p} + \frac{1-f}{R_s}}$$

7.4.4 Easily parallelized problems: The Class NC

In the search for fast parallel problems, a lot of attention has been given to class NC, the class of problems solvable by fast parallel algorithms. A *fast* algorithm is one which runs in polylogarithmic parallel time using a polynomial number of PRAM processors. As with polylogarithmic, we use the term *polynomial* of a quantity to express some powerof that quantity. For example, $n^{1.2}, \sqrt(n), n^5$ are all polynomial expressions of n.

A problem is in class NC if and only if it has an algorithm with

- parallel running time $t(n)$ polylogarithmic on n, and
- uses a polynomial number $p(n)$ of processors.

Example 7 *Sorting n numbers on a PRAM takes $t(n) = O(\log n)$ parallel time using $p(n) = n$ processors. Thus, sorting is in NC. On the other hand, depth-first search (see page 17) is not known if it is.*

Of course, we cannot expect to find NC algorithms for $NP-complete$ problems, i.e. problems for which no polynomial-time sequential algorithm is known. A question that naturally arises is, can we find NC algorithms for all the problems in P (problems solvable by polynomial-time sequential algorithms)? This is the well known $P \overset{?}{=} NC$ question, and it is widely believed that it has a negative answer.

For practical purposes, a work-optimal or work-efficient algorithm is much more useful than one that simply is in NC. However, membership in NC reveals a problem with inherent parallelism and has some theoretical interest as a problem with a promising parallel solution.

HW:It has been demonstrated that it is easier to design algorithms for the more powerful CRCW PRAM model than for the CREW or EREW PRAM models. However, due to the existing simulations between the PRAM models, the notion of efficient parallel algorithms is *robust*, in the sense that any efficient algorithm for some model is still efficient in any weaker model. On the other hand, the notion of optimal algorithms is not robust in this sense.

Maybe define work-efficiency here...

7.4.5 Which metric is more important?

In the last sections we saw that there is a plethora of parallel algorithmic metrics:

1. Parallel Time — the smaller, the better;
2. Number of processors employed — the smaller, the better;
3. Work — the closer to best sequential time, the better;
4. Speedup — the closer to the number of processors employed, the better;
5. Efficiency — the closer to 1, the better,
6. Scalability — desirable property.

It is desirable that an algorithm scores well in each of these metrics. Which one we value more? *It depends on the application we run.* When designing parallel algorithms, we care about work-efficient ones with low running time and processor requirements that is no more than $O(n)$. When implementing an algorithm, we care about good speedup and efficiency, with an eye towards scalability. In real life, however, there is another factor that we have not mentioned yet: The cost of buying and maintaining a machine. Let us see a couple of examples on such an evaluation.

7.5 PRAM abstraction vs. real machines

Even though a few attempts have already been made to construct a real machine that closely simulates a PRAM, it is widely believed that this may not be possible for large number of processors (i.e., more than 64 or so). Then, why bother to use this model?

The PRAM model has many good features that make it attractive when studying parallel algorithms: It hides many of the complexities of real machines, so that the algorithm designer can concentrate on the parallelization of the problem.

What are these hidden details? A PRAM does not explicitly mention any particular interconnection network; whenever two processors need to communicate, they can, without worrying how the message (often called a "packet") will reach the destination. Also, the shared memory feature makes the communication easier; whenever the data of some memory location are needed, a processor can fetch them without worrying which processors memory actually contains the data, or contains a valid copy of them.

However, the same features that make PRAM attractive to algorithm designers, make it unattractive to algorithm implementors. The latter claim that as model it is too abstract to be of any value. Practitioners typically have to deal with today's parallel computers that have comlicated programming environments, slow communication and limited I/O bandwidth. So, when speed is the most important aspect of the program, they have to come up with esoteric code and hairy programming tricks to maximize their gains, and there is no provision in a PRAM for such details.

This dispute is just one of the several ongoing disagreements in parallel computation that have not been settled yet — and is not expected to settle in the near future. All of them, essentially, are over the question *what the perfect, general purpose parallel machine will be like.* But such a machine may never be built: maybe there is no such thing as a perfect parallel machine. Actually, some argue that eventually we will have specialized computers that architecturally will not have many things in common.

The fact, however, remains that the majority of algorithms are being designed for a PRAM, and most of the programs that actually run on existing machines were first designed for a PRAM and then specialized to run faster on the particular platform. These are two of the factors that have contributed to the success of this model so far. In the next section we describe some of the alternative models, for comparison.

7.5.1 *Other Parallel Models

The BSP Model.

The XRAM model was proposed by Valiant as the most realizable parallel model today. In this model, which can optimally simulate EREW and CRCW PRAMs, a usual sequential execution step costs one, but a communication-between-processors step costs g.

An XRAM executes in *supersteps.* Let's assume that within a superstep i some processor executes α_i local operations, sends β_i messages and receives γ_i messages. Then we say that the superstep lasted for

$$r_i = \frac{\alpha_i}{g} + \beta_i + \gamma_i$$

steps. Let $t = \max\{r_i | i \in < p >\}$ where $< p >$ is the set of the processors employed. The run-time of a superstep is $\lceil t/L \rceil \cdot L$ global operations or time $\lceil t/L \rceil \cdot L \cdot g$. The parameter L is usually taken to have value $\lg p$. At the end of L global operations, all the processors know whether a superstep is complete or not. Within each period, however, there is no synchronization among the processors.

The LogP model.

The PPM Model.

The *parallel pointer machine* (PPM) is the parallel version of Knuth's *linking automaton* (called also a *pointer machine*), just as the PRAM is the parallel version of RAM. In the PPM the shared memory can be viewed as a directed graph whose vertices are memory cells, each cell containing some small number of value and pointer fields.

Access of the common storage by some processor is limited to the locations pointed to by the processor's pointer registers. A processor may load a pointer register in one step and perform the usual arithmetic and comparison operations. However, pointer arithmetic is not allowed. As in the PRAM, the communication between processors is done via writing to and reading from the shared memory.

We categorize the PPMs according to how the simultaneous accesses to the shared memory is handled as EREW, CREW and CRCW PPM. Note that these models are weaker than their PRAM counterparts, because a PRAM can easily simulate a PPM but not the other way around, since a PPM does not support array indexing.

7.6 Data-Parallelism?

7.7 Bibliographic Notes

For a description of the sequential RAM, see [**?**]; for the PRAM and its several models, see [**?**, **?**]. The PPM is described in [**?**] as the parallel version of Knuth's linking automaton [**?**]. The BSP is described in [**?**]. The hierarchy of the PRAM models is described in several papers [**?**, **?**, **?**, **?**]. Sorting on a EREW PRAM is described in [**?**, **?**].

The lower bound of computing the OR of n bits on a CREW PRAM is due to Cook, Dwork and Reischuk [**?**]. Simulation between a p-processor

PRIORITY CRCW PRAM and a COMMON CRCW PRAM, is described in [?] and the examples are adapted from [?]. The first $P-complete$ problem was shown in [?]. The $O(\log(n)$ EREW PRAM algorithm is described by [?].

7.8 Exercises

1. A sequential algorithm for solving some particular problem takes $\frac{n^3-n^2}{3}$ steps to complete. When we use a machine with p processors, the time to solve the same problem using a parallel algorithm is $\frac{(n^2-1)(n/2)}{p}$.

   (a) calculate the speedup factor S_p.

   (b) What is the efficiency of the algorithm as $n \rightarrow \infty$?

2. The current prices on some of the (parallel and sequential) computers shown on page 19 are shown below [4]:

Computers	Processors	LinPac (MFlops)	Cost (million $)
Cray C90	16	9700	36
CM-5	512	30400	15
MP-2216	16384	1600	1
KSR-1	32	513	1.5
nCube-2	4	7.5	0.05
Delta	128	2600	3
DEC 3000	1	124	0.05
HP 9000	1	151	0.075
SPARC-10	1	22.4	0.027

   How would you judge the "best" computer? Do you see some alternative to parallel machines that could come from clusters of workstations? Describe clearly your criteria.

3. We claim that "we cannot expect to find NC algorithms for NP-complete problems". Why is this so?

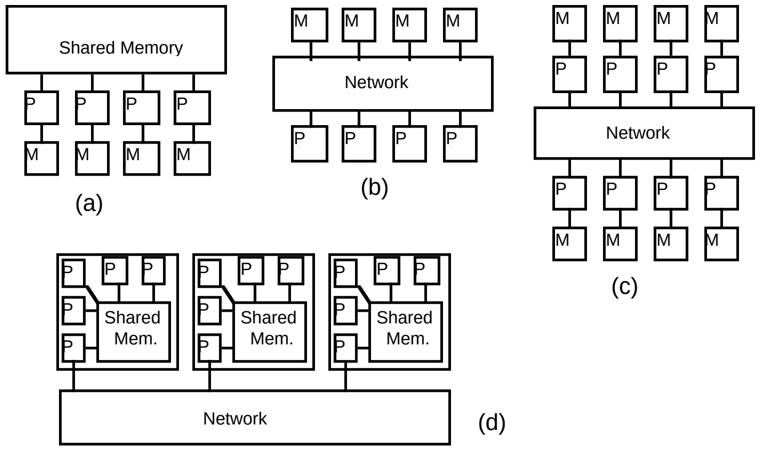

Figure 7.12: Four of the many possible ways to connect processors into a parallel computer: (a) RAMs have been hooked to a large shared memory; (b) CPUs communicate with shared memories through some wires composing an interconnection network; (c) RAMs are connected through a network, no memory is shared here; (d) Chips containing a few processors with their own memory are communicating using a network. A particular processor is used as gateway to the network.

4. (a) Analyze each of the architectures of figure 7.12 (repeated from figure 7.2). How each one of the architecturel designs of figure 7.12 answers the question of section 7.1.1?

   (b) Design two parallel computer architectures (in the spirit of figure 7.12) and analyse them. Can you design something better than the ones shown in the figure?

5. (*) What are the advantages and disadvantages of each of the architectural designs of figure 7.12? Comment on whether some architecture is easy to built, and to be programmed.

6. There is a fourth PRAM model that can be naturally defined, the ERCW, which has not attracted significant interest. Define it and try to explain why it may not be as important.

7. Step 2 of the code used in the simulation of a CRCW PRAM algorithm by a EREW PRAM machine calls for *lexicographic* sorting. Why? What problem may occur if the sorting is not lexicographic? (*Hint:* A sorting algorithm is *stable* if it keeps the relative position of two equally-valued keys after sorting. For example, if in the input (unsorted) array, $A[i] = A[j]$, with $i < j$, then after a stable sorting where $A[i]$ ends up at a location $A[i']$ and $A[j]$ at location $A[j']$, with $i' < j'$.)

8. Theorem 9 shows that we can simulate a PRIORITY CRCW PRAM algorithm on a COMMON CRCW PRAM machine without slowdown, using $O(r^2)$ additional processors. Theorem 8 implies that we could also do it with no additional processors, but it would slow down the simulation by a factor of $O(\log r)$. Argue about the pros and cons

[4]These data were posted on June 93 on the network

of each method. Describe situations where each method would be preferable.

9. Fact 1 uses the word “polynomially” to give an upper bound on the number of processors that could be used to compute the OR function. Why? What happens when exponentially many processors are available?