14. Parallel Computing

14.1 Introduction

This chapter describes approaches to problems to which multiple computing agents are applied simultaneously.

By "parallel computing", we mean using several computing agents concurrently to achieve a common result. Another term used for this meaning is "concurrency". We will use the terms *parallelism* and *concurrency* synonymously in this book, although some authors differentiate them.

Some of the issues to be addressed are:

> What is the role of parallelism in providing clear decomposition of problems into sub-problems?
>
> How is parallelism specified in computation?
>
> How is parallelism effected in computation?
>
> Is parallel processing worth the extra effort?

14.2 Independent Parallelism

Undoubtedly the simplest form of parallelism entails computing with totally independent tasks, i.e. there is no need for these tasks to communicate. Imagine that there is a large field to be plowed. It takes a certain amount of time to plow the field with one tractor. If two equal tractors are available, along with equally capable personnel to man them, then the field can be plowed in about half the time. The field can be divided in half initially and each tractor given half the field to plow. One tractor doesn't get into another's way if they are plowing disjoint halves of the field. Thus they don't need to communicate. Note however that there is some initial overhead that was not present with the one-tractor model, namely the need to divide the field. This takes some measurements and might not be that trivial. In fact, if the field is relatively small, the time to do the divisions might be more than the time saved by the second tractor. Such overhead is one source of dilution of the effect of parallelism.

Rather than dividing the field only two ways, if N tractors are available, it can be divided N ways, to achieve close to an N-fold gain in speed, if overhead is ignored. However, the larger we make N, the more significant the overhead becomes (and the more tractors we have to buy).

In UNIX® command-line shells, independent parallelism can be achieved by the user within the command line. If c_1, c_2, ..., c_n are commands, then these commands can be executed in parallel in principal by the compound command:

c_1 & c_2 & .... & c_n

This does not imply that there are *n* processors that do the work in a truly simultaneous fashion. It only says that logically these commands can be done in parallel. It is up to the operating system to allocate processors to the commands. As long as the commands do not have interfering side-effects, it doesn't matter how many processors there are or in what order the commands are selected. If there are interfering side-effects, then the result of the compound command is not guaranteed. This is known as indeterminacy, and will be discussed in a later section.

There is a counterpart to independent parallelism that can be expressed in C++. This uses the *fork* system call. Execution of `fork()` creates a new *process* (program in execution) that is executing the same code as the program that created it. The new process is called a *child*, with the process executing `fork` being called the *parent*. The child gets a complete, but independent, copy of all the data accessible by the parent process.

When a child is created using fork, it comes to life as if it had just completed the call to fork itself. The only way the child can distinguish itself from its parent is by the return value of fork. The child process gets a return value of 0, while the parent gets a non-zero value. Thus a process can tell whether it is the parent or the child by examining the return value of fork. As a consequence, the program can be written so as to have the parent and child do entirely different things within the same program that they share.

The value returned by fork to the parent is known as the *process id* (pid) of the child. This can be used by the parent to control the child in various ways. One of the uses of the pid, for example, is to identify that the child has terminated. The system call wait, when given the pid of the child, will wait for the child to terminate, then return. This provides a mechanism for the parent to make sure that something has been done before continuing. A more liberal mechanism involves giving wait an argument of 0. In this case, the parent waits for termination of any one of its children and returns the pid of the first that terminates.

Below is a simple example that demonstrates the creation of a child process using fork in a UNIX® environment. This is C++ code, rather than Java, but hopefully it is close enough to being recognizable that the idea is conveyed.

```
#include <iostream.h>                    // for <<, cin, cout,
cerr
#include <sys/types.h>                   // for pid_t
#include <unistd.h>                      // for fork()
#include <wait.h>                        // for wait()
```

```
main()
{
pid_t pid, ret_pid;  // pids of forked and finishing child

pid = fork();                              // do it

// fork() creates a child that is a copy of the parent
// fork() returns:
//       0 to the child
//       the pid of the child to the parent

if( pid == 0 )
  {
  // If I am here, then I am the child process.
  cout << "Hello from the child" << endl << endl;
  }
else
  {
  // If I am here, then I am the parent process.
  cout << "Hello from the parent" << endl << endl;

  ret_pid = wait(0);                       // wait for the child
  if( pid == ret_pid )
    cout << "child pid matched" << endl << endl;
  else
    cout << "child pid did not match" << endl << endl;
  }
}
```

The result of executing this program is:

```
Hello from the parent

Hello from the child

child pid matched
```

14.3 Scheduling

In the UNIX® example above, there could well be more processes than there are processors to execute those processes. In this case, the *states* of non-running processes are saved in a pool and executed when a processor becomes available. This happens, for example, when a process terminates. But it also happens when a process waits for i/o. This notion of *multiprocessing* is one of the key ways of keeping a processor busy when processes do i/o requests.

In the absence of any priority discipline, a process is taken from the pool whenever a processor becomes idle. To provide an analogy with the field-plowing problem, work apportionment is simplified in the following way: Instead of dividing the field into one segment per tractor, divide it into many small parcels. When a tractor is done plowing its current parcel, it finds another unplowed parcel and does that. This scheme, sometimes called *self-scheduling*, has the advantage that the tractors stay busy even if some run at much different rates than others. The opposite of self-scheduling, where the work is divided up in advance, will be called *a priori scheduling*.

14.4 Stream Parallelism

Sets of tasks that are totally independent of each other do not occur as frequently as one might wish. More interesting are sets of tasks that need to communicate with each other in some fashion. We already discussed command-line versions of a form of communication in the chapter on high-level functional programming. There, the compound command

$c_1 \mid c_2 \mid .... \mid c_n$

is similar in execution to the command with | replaced by &, as discussed earlier. However, in the new case, the processes are synchronized so that the input of one waits for the output of another, on a character-by-character basis, as communicated through the standard inputs and standard outputs. This form of parallelism is called *stream parallelism*, suggesting data flowing through the commands in a stream-like fashion.

The following C++ program shows how pipes can be used with the C++ iostream interface. Here class filebuf is used to connect file buffers that are used with streams to the system-wide filed descriptors that are produced by the call to function `pipe`.

```
#include <streambuf.h>
#include <iostream.h>          // for <<, cin, cout, cerr
#include <stdio.h>             // for fscanf, fprintf
#include <sys/types.h>         // for pid_t
#include <unistd.h>            // for fork()
#include <wait.h>              // for wait()

main()
{
int pipe_fd[2];                // pipe file descriptors:
                               // pipe_fd[0] is used for the read end
                               // pipe_fd[1] is used for the write end

pid_t pid;                     // process id of forked child

pipe(pipe_fd);                 // make pipe, set file descriptors

pid = fork();                  // fork a child

int chars_read;

if( pid == 0 )
  {
  // Child process does this

  // read pipe into character buffer repeatedly, until end-of-file,
  // sending what was read to cout

  // note: a copy of the array pipe_fd exists in both processes

  close(pipe_fd[1]);          // close unused write end

  filebuf fb_in(pipe_fd[0]);
  istream in(&fb_in);

  char c;

  while( in.get(c) )
    cout.put(c);

  cout << endl;
  }
else
  {
  close(pipe_fd[0]);           // close unused read end

  filebuf fb_out(pipe_fd[1]);
  ostream out(&fb_out);

  char c;

  while( cin.get(c) )
    out.put(c);

  close(pipe_fd[1]);

  wait(0);                    // wait for child to finish
  }
}
```

It is difficult to present a plowing analogy for stream parallelism – this would be like the field being forced through the plow. A better analogy would be an assembly line in a factory. The partially-assembled objects move from station to station; the parallelism is among the stations themselves.

Exercises

Which of the following familiar parallel enterprises use self-scheduling, which use *a priori* scheduling, and which use stream parallelism? (The answers may be locality-dependent.)

1 • check-out lanes in a supermarket

2 • teller lines in a bank

3 • gas station pumps

4 • tables in a restaurant

5 • carwash

6 •••• Develop a C++ class for pipe streams that hides the details of file descriptors, etc.

14.5 Expression Evaluation

Parallelism can be exhibited in many kinds of expression evaluations. The UNIX® command-line expressions are indeed a form of this. But how about with ordinary arithmetic expressions, such as

(a + b) * ((c - d) / (e + f)))

Here too there is the implicit possibility of parallelism, but at a finer level of granularity. The sum a + b can be done in parallel with the expression (c - d) / (e + f). This expression itself has implicit parallelism. Synchronization is required in the sense that the result of a given sub-expression can't be computed before the principal components have been computed. Interestingly, this form of parallelism shows up if we inspect the dag (directed acyclic graph) representation of the expression. When there are two nodes with neither having a path to the other, the nodes could be done concurrently in principle.

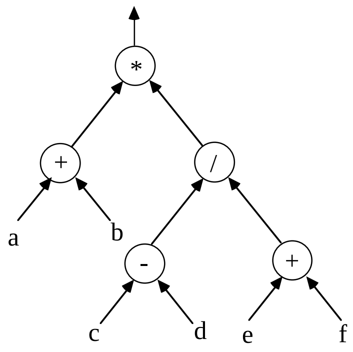

Figure 297: Parallelism in an expression tree: The left + node can be done in parallel with any of the nodes other than the * node. The - node can be done in parallel with the right + node.

In the sense that synchronization has to be done from two different sources, this form of parallelism is more complex than stream parallelism. However, stream parallelism has the element of repeated synchronization (for each character) that scalar arithmetic expressions do not. Still, there is a class of languages in which the above expression might represent computation on vectors of values. These afford the use of stream parallelism in handling the vectors.

For scalar arithmetic expressions, the level of granularity is too fine to create processes – the overhead of creation would be too great compared to the gain from doing the operations. Instead, arithmetic expressions that can be done in parallel are usually used to exploit the "pipelining" capability present in high-performance processors. There are typically several instructions in execution simultaneously, at different stages. A high degree of parallelism translates into lessened constraints among these instructions, allowing more of the instruction-execution capabilities to be in use simultaneously.

Expression-based parallelism also occurs when data structures, such as lists and trees, are involved. One way to exploit a large degree of parallelism is through the application of functions such as *map* on large lists. In mapping a function over list, we essentially are specifying one function application for each element of the list. Each of these applications is independent of the other. The only synchronization needed is in the use vs. formation of the list itself: a list element can't be used before the corresponding application that created it is done.

Recall that the definition of *map* in rex is:

```
map(Fun, []) => [].

map(Fun, [A | X]) => [apply(Fun, A) | map(Fun, X)].
```

The following figure shows how concurrency results from an application of *map* of a function *f* to a list [x_1, x_2, x_3, ...]. The corresponding function in rex that evaluates those function applications in parallel is called *pmap*.

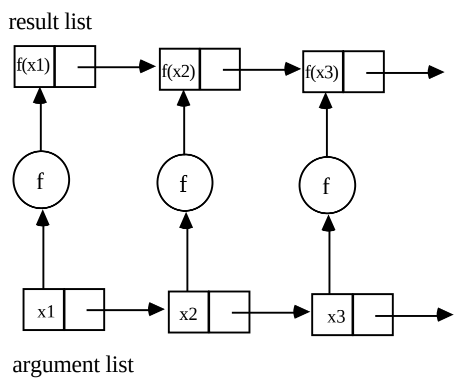

Figure 298: Parallel mapping a function over a list for independent execution. Each copy of f can be simultaneously executing.

Exercises

Which of the following programs can exploit parallelism for improved performance over sequential execution? Informally describe how.

1 •• finding the maximum element in an array

2 •• finding an element in a sorted array

3 •• merge sort

4 •• insertion sort

5 •• Quicksort

6 ••• finding a maximum in a uni-modal array (an array in which the elements are increasing up to a point, then decreasing)

7 •• finding the inner-product of two vectors

8 •• multiplying two matrices

14.6 Data-Parallel Machines

We next turn our attention to the realization of parallelism on actual computers. There are two general classes of machines for this purpose: data-parallel machines and control-parallel machines. Each of these classes can be sub-divided further. Furthermore, each class of machines can simulate the other, although one kind of machine will usually be preferred for a given kind of problem.

Data parallel machines can be broadly classified into the following:

SIMD multiprocessors
Vector Processors
Cellular Automata

SIMD Multiprocessors

"SIMD" stands for "single-instruction stream, multiple data stream". This means that there is one stream of instructions controlling the overall operation of the machine, but multiple data operation units to carry out the instructions on distinct data. The general structure of a SIMD machine can thus be depicted as follows:

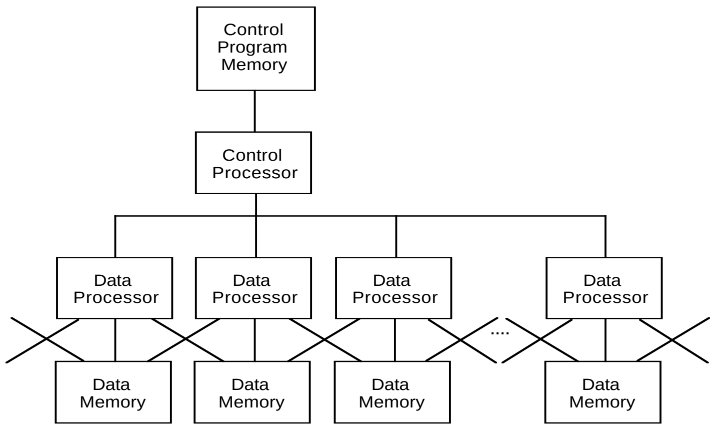

Figure 299: SIMD multiprocessor organization

Notice that in the SIMD organization, each data processor is coupled with its own data memory. However, in order to get data from one memory to another, it is necessary for each processor to have access to the memories of its neighbors, at a minimum. Without this, the machine would be reduced to a collection of almost-independent computers.

Also not clear in the diagram is how branching (jumps) dependent on data values take place in control memory. There must be some way for the control processor to look at selected data to make a branch decision. This can be accomplished by instructions that form some sort of aggregate (such as the maximum) from data values in each data processors' registers.

SIMD Multiprocessors are also sometimes called *array processors*, due to their obvious application to problems in which an array can be distributed across the data memories.

Vector Processors

The name "vector processor" sounds similar to "array processor" discussed above. However, vector processor connotes something different to those in the field: a processor in which the data stored in *registers* are vectors. Typically there are both scalar registers and vector registers, with the instruction code determining whether an addressed register is one or the other. Vector processors differ from array processors in that not all elements of the vector are operated on concurrently. Instead pipelining is used to reduce the cost of a machine that might otherwise have one arithmetic unit for each vector element.

Typically, vector operations are floating-point. Floating point arithmetic can be broken into four to eight separate stages. This means that a degree of concurrency equal to the number of stages can be achieved without additional arithmetic units. If still greater concurrency is desired, additional pipelined arithmetic units can be added, and the vectors apportioned between them.

Cellular Automata

A cellular automaton is a theoretical model that, in some ways, is the ultimate data parallel machine. The machine consists of an infinite array of cells. Each cell contains a state drawn from a finite set, as well as a finite state machine, which is typically the same for every cell. The state transitions of the machine use the cell's own state, as well as the states of selected other cells (known as the cell's neighbors), to determine the cell's next state. All cells compute their next states simultaneously. The entire infinite array therefore operates in locked-step fashion.

In most problems of interest, only a finite number of the cells are in a non-quiescent state. In other words, most of the cells are marking time. The state-transition rules are usually designed this way. The automaton is started with some specified cells being non-quiescent. This is how the input is specified. Then non-quiescent states generally propagate from those initial non-quiescent cells.

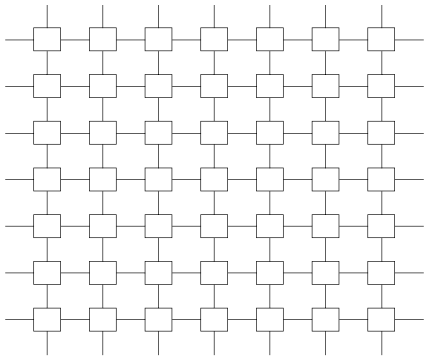

Figure 300: Fragment of a two-dimensional cellular automaton; cells extend forever in all four compass directions

We have shown above a two-dimensional cellular automaton. However, cellular automata can be constructed in any number of dimensions, including just one. It is also possible to consider irregular cellular automata, connected as an arbitrary graph structure, in which there is no clear notion of dimension. Most cellular automata that have been studied are rectangular ones in one, two, or three dimensions. It is also possible to use different sets of neighbors than the one shown. For example, an eight-neighbor automaton is common.

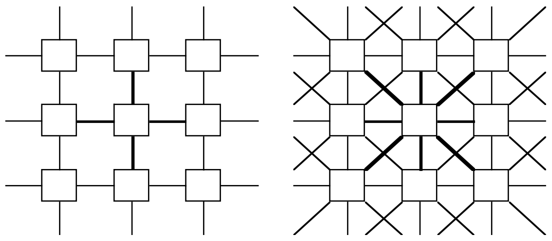

Figure 301: Four- vs. eight-neighbor cellular automata

Cellular automata were studied early by John von Neumann. He showed how Turing machines can be embedded within them, and moreover how they can be made to reproduce themselves. A popular cellular automaton is Conway's "Game of Life". Life is a two-dimensional cellular automaton in which each cell has eight neighbors and only two states (say "living" and "non-living", or simply 1 and 0). The non-living state corresponds to the quiescent state described earlier.

The transition rules for Life are very simple: If three of a cell's neighbors are living, the cell itself becomes living. Also, if a cell is living, then if its number of living neighbors is other than two or three, it becomes non-living.

The following diagram suggests the two kinds of cases in which the cell in the center is living in the next state. Each of these rules is only an example. A complete set of rules would number sixteen.

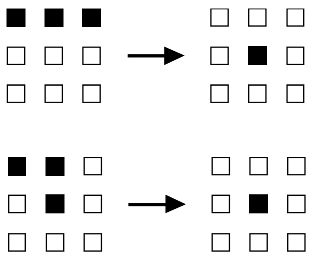

Figure 302: Examples of Life rules

Many interesting phenomena have been observed in the game of life, including patterns of living cells that appear to move or "glide" through the cellular space, as well as patterns of cells that produce these "gliders".

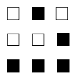

Figure 303: A Life glider pattern

It has been shown that Life can simulate logic circuits (using the presence or absence of a glider to represent a 1 or 0 "flowing" on a wire). It has also been shown that Life can simulate a Turing machine, and therefore is a universal computational model.

Exercises

1 • Enumerate all of the life rules for making the center cell living.

2 •• Write a program to simulate an approximation to Life on a bounded grid. Assume that cells outside the grid are forever non-living.

3 •• In the Fredkin automaton, there are eight neighbors, as with life. The rules are that a cell becomes living if it is non-living and an odd number of neighbors are living. It becomes non-living if it is living and an even number of neighbors are living. Otherwise it stays as is. Construct a program that simulates a Fredkin automaton. Observe the remarkable property that any pattern in such an automaton will reproduce itself in sufficient time.

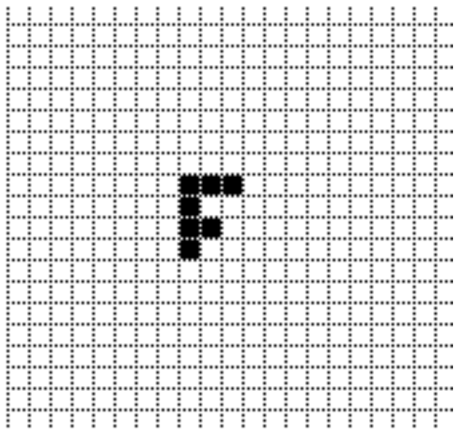

Figure 304: The Fredkin automaton, with an initial pattern

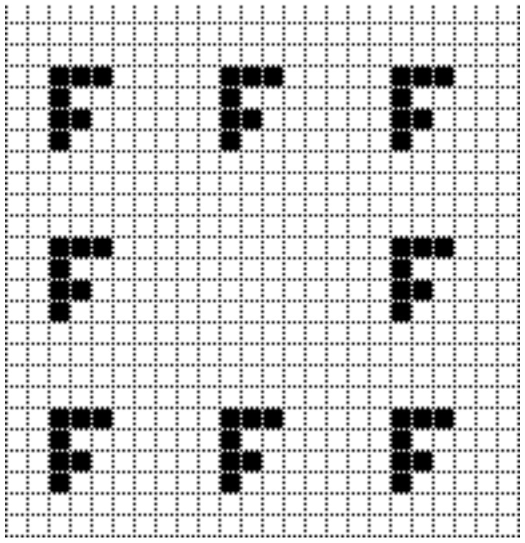

Figure 305: The Fredkin automaton, after 8 transitions from the initial pattern

4 ••• For the Fredkin automaton, make a conjecture about the reproduction time as a function of the number of cells to be reproduced.

5 •• Show that any Turing machine can be simulated by an appropriate cellular automaton.

6 •••• Write a program to simulate Life on an unbounded grid. This program will have to dynamically allocate storage when cells become living that were never living before.

7 ••••• A "garden-of-Eden" pattern for a cellular automaton is one that cannot be the successor of any other state. Find such a pattern for Life.

14.7 Control-Parallel Machines

We saw earlier how data-parallel machines rely on multiple processors conducting similar operations on each of a large set of data. In control-parallel machines, there are multiple instruction streams and no common control. Thus these machines are also often called MIMD ("Multiple-Instruction, Multiple-Data") machines. Within this category, there are two predominant organizations:

Shared memory

Distributed memory

Shared Memory Multiprocessors

In a shared memory machine, all processors conceptually share a common memory. Each processor is executing its own instruction stream. The processors can communicate with one another based on inspection and modification of data common to both of them.

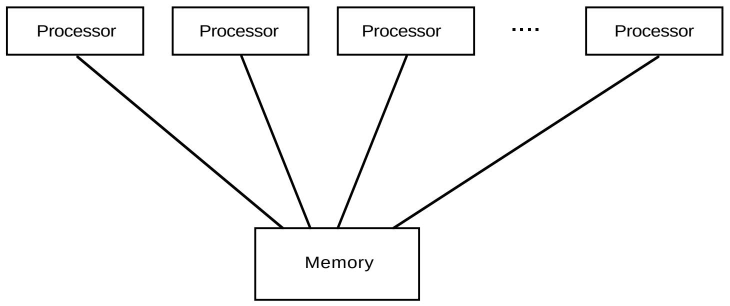

Figure 306: Shared Memory Multiprocessor

As an example of how shared memory permits one process to control another, suppose we want to make one process wait until an event occurs, as determined by another process. We can achieve this in a shared memory machine by agreeing on a common location to contain a flag bit. We agree in advance that a 0 in this bit means the event has not occurred, while a 1 indicates it has. Then the process that potentially waits will test the bit. If it is 0, it loops back and tests again, and so on. If it is 1, it continues. All the signaling processor has to do is set the bit to 1.

The following code fragments show the use of a flag for signaling between two processes.

Flag = 0 initially

```
Process A:                       Process B:

    A1:  ....                    B1: ....
    A2: Flag = 1;                B2: if( Flag == 0 )
                                        goto B2;
    A3: ....                     B3: ....
```

The following state diagram shows that the signaling scheme works. The progress of process A corresponds to the vertical dimension, and that of process B to the horizontal. The components of each state are:

(ip of A, ip of B, Flag)

Note that states in which ip of B is B3 cannot be reached as long as Flag is 0. Only when A has set the flag to 1 can such a state (at the lower-left corner) be reached.

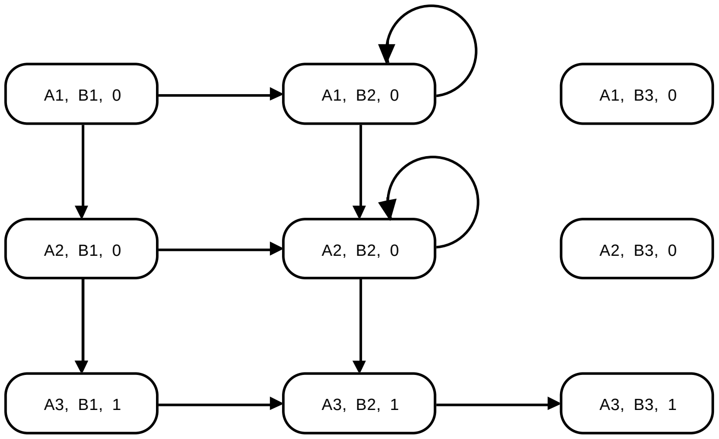

Figure 307: State-transition diagram of multiprocessor flag signaling

The type of signaling described above is called "busy-waiting" because the looping processor is kept busy, but does no real work. To avoid this apparent waste of the processor as a resource, we can interleave the testing of the bit with some useful work. Or we can have the process "go to sleep" and try again later. Going to sleep means that the process gives up the processor to another process, so that more use is made of the resource.

Above we have shown how a shared variable can be used to control the progress of one process in relation to another. While this type of technique could be regarded as a feature of shared memory program, it can also present a liability. When two or more processors share memory, it is possible for the overall result of a computation to fail to be unique. This phenomenon is known as indeterminacy. Consider, for example, the case where two processors are responsible for adding up a set of numbers. One way to do this would be to share a variable used to collect the sum. Depending on how this variable is handled, indeterminacy could result. Suppose, for example, that a process assigns the sum variable to a local variable, adds a quantity to the local variable, then writes back the result to the sum:

```
local = sum;
local += quantity to be added;
sum = local;
```

Suppose two processes follow this protocol, and each has its own local variable. Suppose that sum is initially 0 and one process is going to add 4 and the other 5. Then depending on how the individual instructions are interleaved, the result could be 9 or it could be 4 or 5. This can be demonstrated by a state diagram. The combined programs are

```
A1:  localA = sum;              B1: localB = sum;
A2:  localA += 4;               B2: localA += 5;
A3:  sum = localA;              B3: sum = localB;
A4:                             B4:
```

The state is:

(ip of A, ip of B, localA, localB, sum)

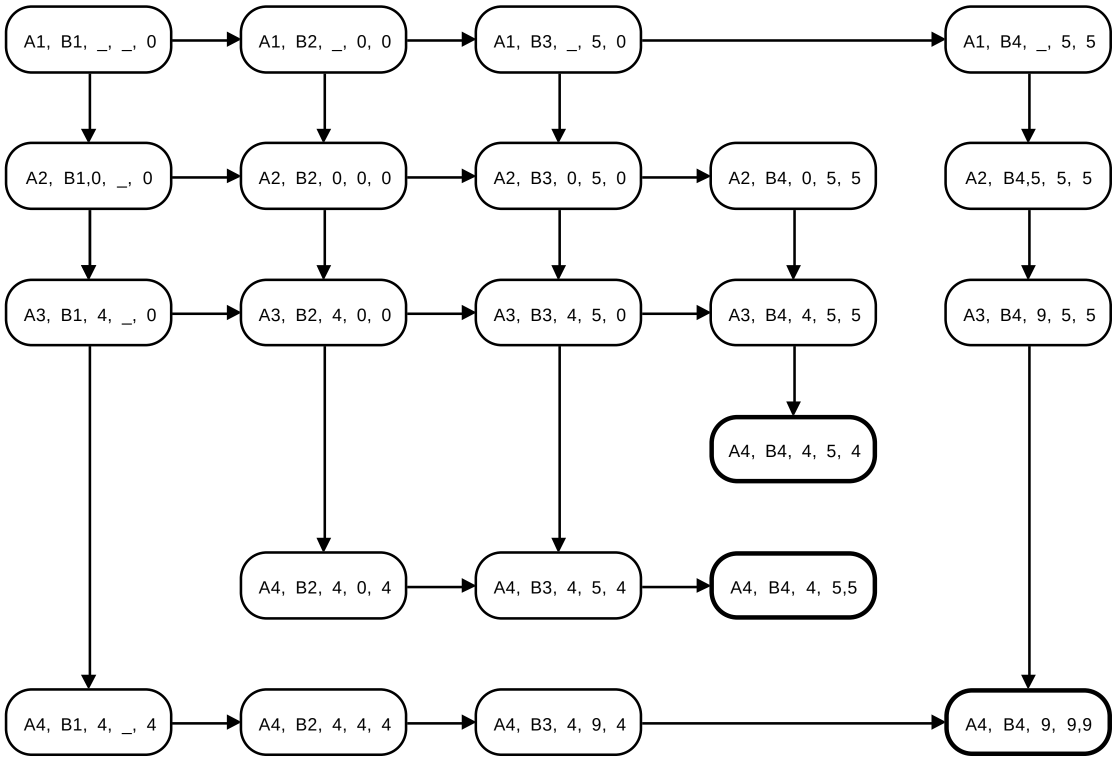

Figure 308: A state diagram exhibiting indeterminacy

Note that in the state diagram there are three states having the ip components A4, B4, each with a different value for sum. One state corresponds to the two processes having done their operations one after the other, and the other two corresponds to one process writing while the other is still computing. Thus we have *indeterminacy* in the final value of sum. This phenomenon is also called a *race condition*, as if processes A and B were racing with each other to get access to the shared variable.

Race conditions and indeterminacy are generally undesirable because they make it difficult to reason about parallel programs and to prove their correctness. Various abstractions and programming techniques can be used to reduce the possibilities for indeterminacies. For example, if we use a purely functional programming model, there are no shared variables and no procedural side-effects. Yet there can still be a substantial amount of parallelism, as seen in previous examples, such as the function *map*.

Semaphores

Computer scientists have worked extensively on the problem of using processor resources efficiently. They have invented various abstractions to not only allow a process to go to sleep, but also to wake it when the bit has been set, and not sooner. One such abstraction is known as a semaphore. In one form, a semaphore is an object that contains a positive integer value. The two methods for this object are called P and V. When P is done on the semaphore, if the integer is positive, it is lowered by one (P is from a Dutch word meaning "to lower") and the process goes on. If it is not positive, however, the process is put to sleep until a state is reached in which lower can be done without making the integer negative. The only way that such a state is reached is by another process executing the V ("to raise") operation. If no process is sleeping for the semaphore, the V operation simply increments the integer value by one. If, on the other hand, at least one process is sleeping, one of those processes is chosen for awakening and the integer value remains the same (i.e. the net effect is as if the value had been raised and then lowered, the latter by the sleeping process that was not able to lower it earlier). The exact order for awakening is not specified. Most often, the process sleeping for the longest time is awakened next, i.e. a queue data structure is used.

The following program fragments show the use of a semaphore for signaling.

Semaphore S's integer value is 0 initially

Process A:

```
A1:  ....
A2: V(S);
A3: ....
```

Process B:

```
B1: ....
B2: P(S)
B3: ....
```

The state-transition diagram is similar to the case using an integer flag. The main difference is that no looping is shown. If the semaphore value is 0, process B simply cannot proceed. The components of each state are:

(ip of A, ip of B, Semaphore value)

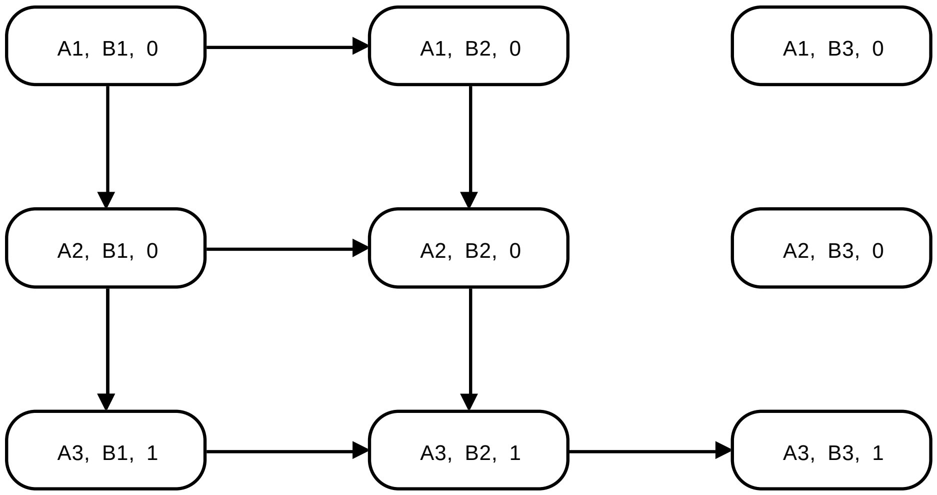

Figure 309: State diagram for signaling using a semaphore

Above we saw a flag and a semaphore being used to *synchronize* one process to another. Another type of control that is often needed is called *mutual exclusion*. In contrast to synchronization, this form is symmetric. Imagine that there is some data to which two processes have access. Either process can access the data, but only one can access it at a time. The segment of code in which a process accesses the data is called a *critical section*. A semaphore, initialized to 1 rather than 0, can be used to achieve the mutual exclusion effect.

Semaphore S's integer value is 1 initially

Process A: Process B:

```
A1: ....                        B1: ....
A2: P(S);                       B2: P(S)
A3: critical section            B3: critical section
A4: V(S)                        B4: V(S)
A5: ....                        B5: ....
```

A useful extension of the semaphore concept is that of a *message queue* or *mailbox*. In this abstraction, what was a non-negative integer in the case of the semaphore is replaced by a queue of messages. In effect, the semaphore value is like the length of the queue. A process can send a message to another through a common mailbox. For example, we can extend the P operation, which formerly waited for the semaphore to have a positive value, to return the next message on the queue:

P(S, M); sets M to the next message in S

If there is no message in S when this operation is attempted, the process doing P will wait until there is a message. Likewise, we extend the V operation to deposit a message:

V(S, M);   puts message M into S

Mailboxes between UNIX® processes can be simulated by an abstraction known as a *pipe*. A pipe is accessed in the same way a file is accessed, except that the pipe is not really a permanent file. Instead, it is just a buffer that allows bytes to be transferred from one process to another in a disciplined way. As with an input stream, if the pipe is currently empty, the reading process will wait until something is in the pipe. In UNIX®, a single table common to all processes is used to hold descriptors for open files. Pipes are also stored in this table. A pipe is created by a system call that returns two file descriptors, one for each end of the pipe. The user of streams in C++ does not typically see the file descriptors. These are created dynamically and held in the state of the stream object. By appropriate low-level coding, it is possible to make a stream object connect to a pipe instead of a file.

Exercises

1 •• Construct a state-transition diagram for the case of a semaphore used to achieve mutual exclusion. Observe that no state is reached in which both processes are in their critical sections.

2 ••• Show that a message queue can be constructed out of ordinary semaphores. (Hint: Semaphores can be used in at least two ways: for synchronization and for mutual exclusion.)

3 ••• Using a linked-list to implement a message queue, give some examples of what can go wrong if the access to the queue is not treated as a critical section.

14.8 Distributed Memory Multiprocessors

A distributed memory multiprocessor provides an alternative to shared memory. In this type of design, processors don't contend for a common memory. Instead, each processor is closely linked with its own memory. Processors must communicate by sending *messages* to one another. The only means for a processor to get something from another processor's memory is for the former to send a message to the latter indicating its desire. It is up to the receiving processor to package a response as a message back to the requesting processor.

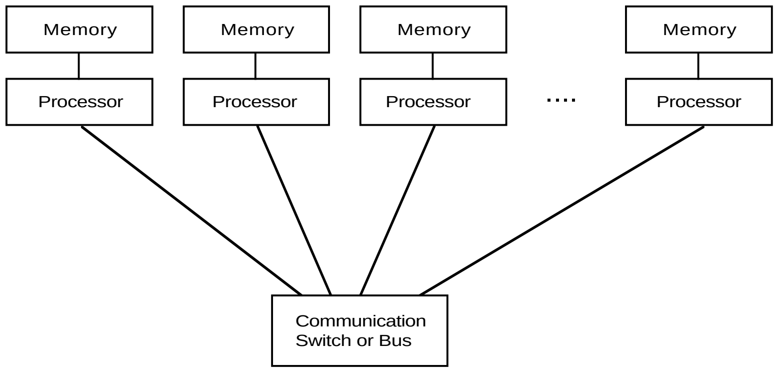

Figure 310: Distributed Memory Multiprocessor

What possible advantage could a distributed memory multiprocessor have? For one thing, there is no contention for a common memory module. This improves performance. On the other hand, having to go through an intermediary to get information is generally slower than in the case of shared memory.

A related issue is known as *scalability*. Assuming that there is enough parallelism inherent in the application, one might wish to put a very large number of processors to work. Doing this exacerbates the problem of memory contention: only one processor can access a single memory module at a time. The problem can be alleviated somewhat by adding more memory modules to the shared memory configuration. The bottleneck then shifts to the processor-memory communication mechanism. If, for example, a bus interconnection is used, there is a limit to the number of memory transactions per unit time, as determined by the bus technology. If the number of processors and memory modules greatly exceeds this limit, there will be little point in having multiple units. To overcome the bus limitation, a large variety of multi-stage interconnect switches have been proposed. A typical example of such a switch is the *butterfly* interconnection, as shown below.

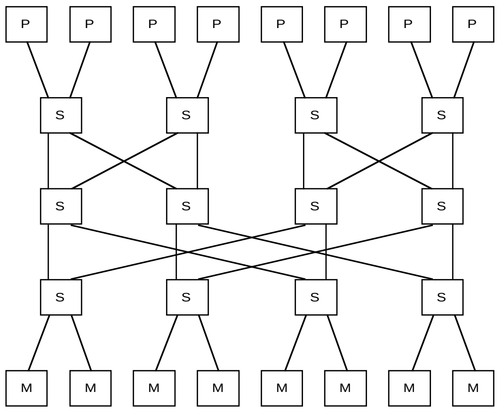

Figure 311: Butterfly interconnection for shared memory

A major problem with such interconnection schemes is known as *cache coherency*. If processors cache data in memory, there is the possibility that one processor could have a datum in its cache while another processor updates the memory version of that datum. In order to manage this possibility, a cached copy would have to be invalidated whenever such an update occurs. Achieving this invalidation requires communication from the memory to the processor, or some kind of vigilance (called "snooping") in the interconnection switch. On the other hand, a distributed memory computer does not incur the cache coherency problem, but rather trades this for a generally longer access time for remote data.

Distributed memory multiprocessors bet on a high degree of *locality*, in the sense that most accesses will be made to local memory, in order to achieve performance. They also mask latency of memory accesses by switching to a different thread when a remote access is needed.

Networking and Client-Server Parallelism

Another type of distributed memory computer is a *computer network*. In this type of system, several computers, each capable of operating stand-alone, are interconnected. The interconnection scheme is typically in the nature of a bus, such as an Ethernet®. These networks were not originally conceived for purposes of parallel computing as much as they were for information sharing. However, they can be used for parallel computing effectively if the degree of locality is very high.

A common paradigm for parallel computing in a network is known as *client-server*. In this model, long-term processes running on selected nodes of the network provide a service for processes on other nodes. The latter, called *clients*, communicate with the server by sending it messages. Many servers and clients can be running concurrently. A given node might support both client and server processes. Furthermore, a server of one function might be a client of others. The idea is similar to object-oriented computing, except that the objects in this case run concurrently with each other, rather than being dormant until the next message is received.

From the programmer's point of view, communication between client and server takes place using data abstractions such as *sockets*. The socket concept permits establishment of a connection between a client and a server by the client knowing the server's address in the network. Knowing the address allows the client to send the server an initial connect message. After connection is established, messages can be exchanged without the need, by the program, to use the address explicitly. Of course, the address is still used implicitly to route the message from one node to another. A common form of socket is called the *stream socket*. In this form, once the connection is established, reading and writing appears to be an ordinary i/o operation, similar to using streams in C++.

14.9 Speedup and Efficiency

The *speedup* of a parallel computation is the ratio of the sequential execution time to the parallel execution time for the same problem. Conventions vary as to whether the same algorithm has to be used in both time computaions. The most fair comparison is probably to compare the speed of the parallel algorithm to that of the best-known sequential algorithm.

An ideal speedup factor would be equal to the number of processors, but rarely is this achieved. The reasons that prevent it from being achieved are: (i) not all problems lend themselves to parallel execution; some are inherently sequential, and (ii) there is overhead involved in creating parallel tasks and communicating between them.

The idea of efficiency attempts to measure overhead; it is defined as the amount of work actually done in a parallel computation divided by the product of the run-time and the number of processors, the latter product being regarded as the *effort*

A thorough study of speedup issues is beyond the scope of this book. Suffice it to say that difficulty in attaining acceptable speedup on a large class of problems has been one of the main factors in the slow acceptance of parallel computation. The other factor is the extra programming effort typically required to achieve parallel execution. The next chapter mentions Amdahl's law, which is one attempt at quantifying an important issue, namely that some algorithms might be inherently sequential.

14.9 Chapter Review

Define the following terms:

cellular automaton
client-server
distributed-memory
efficiency
expression parallelism
fork
map
MIMD
pipeline processing
process
scheduling
shared-memory
semaphore
SIMD
speedup
stream-parallelism
vector processor

14.10 Further Reading

Elwyn Berlekamp, John Conway, and Richard Guy. *Winning ways for your mathematical plays*, vol. 2, Academic Press, 1982. [Chapter 25 discusses the game of Life, including self-reproduction and simulation of switching circuits. Moderate.]

A.W. Burks (ed.), *Essays on cellular automata*, University of Illinois Press, 1970. [A collection of essays by Burks, von Neumann, Thatcher, Holland, and others. Includes an analysis of von Neumann's theory of self-reproducing automata. Moderate.]

Vipin Kumar, et al., *Introduction to parallel computing – Design and Analysis of Algorithms*, Benjamin/Cummings, 1994. [Moderate.]

Michael J. Quinn, *Parallel computing –theory and practice*, McGraw-Hill, 1994. [An introduction, with emphasis on algorithms. Moderate.]

William Poundstone, *The recursive universe*, Contemporary books, Inc., Chicago, Ill., 1985. [A popular discussion of the game of Life and physics. Easy.]

Evan Tick, *Parallel logic programming*, MIT Press, 1991. [Ties together parallelism and logic programming. Moderate.]

Stephen Wolfram, Cellular automata and complexity, Addison-Wesley, 1994. [A collection of papers by Wolfram. Difficult.]