Introduction to Parallel Processing

Norman Matloff
Department of Computer Science
University of California at Davis

March 28, 2006

Contents

1 Overview

There is an ever-increasing appetite among computer users for faster and faster machines. This was epitomized in a statement by Steve Jobs, founder/CEO of Apple and Pixar. He noted that when he was at Apple in the 1980s, he was always worried that some other company would come out with a faster machine than his. But now at Pixar, whose graphics work requires extremely fast computers, he is always hoping someone produces faster machines, so that he can use them!

A major source of speedup is the parallelizing of operations. Parallel operations can be either within-processor, such as with pipelining or having several ALUs within a processor, or between-processor, in which many processor work on different parts of a problem in parallel. Our focus here is on between-processor operations.

For example, the Registrar's Office at UC Davis uses shared-memory multiprocessors for processing its on-line registration work. A shared-memory multiprocessor machine consists of several processors, plus a lot of memory, all connected to the same bus or other interconnect. All processors then access the same memory chips. As of March 2004, the Registrar's current system was a SPARC Sunfire 3800, with 16 GB RAM and eight 900 MHz UltraSPARC III+ CPUs.[1]

Online registration involves an enormous amount of database computation. In order to handle this computation reasonably quickly, the program partitions the work to be done, assigning different portions of the database to different processors. Currently database work comprises the biggest use of shared-memory parallel machines. It is due to the database field that such machines are now so successful commercially.

As the Pixar example shows, highly computation-intensive applications like computer graphics also have a need for these fast parallel computers. No one wants to wait hours just to generate a single image, and the use of parallel processing machines can speed things up considerably. For example, consider **ray tracing** operations. Here our code follows the path of a ray of light in a scene, accounting for reflection and absorbtion of the light by various objects. Suppose the image is to consist of 1,000 rows of pixels, with 1,000 pixels per row. In order to attack this problem in a parallel processing manner with, say, 25 processors, we could divide the image into 25 squares of size 200x200, and have each processor do the computations for its square.[2]

Parallel processing machines tend to be quite expensive, but another popular platform for parallel processing which is much cheaper is a network of workstations (NOW). Here the hardware consists of a set of computers which happen to be connected via a network, such as in our Computer Science Instructional Facility (CSIF) at UCD.[3] Most of the time these machines are used independently, with the role of the network being for e-

[1] You might consider 900 MHz somewhat slow, ironic for a machine whose goal is speed. But speeds for RISC chips such as the SPARC cannot be easily compared with those of CISC chips like Intel. Moreover, these systems have especially fast I/O, which is crucial for database applications. That, together with the fact that we have many processors working in parallel, does indeed make for a very fast machine.

It is true, though, that Sun probably has faster SPARCs than 900 MHz. The problem is that in a multiprocessor system, the processors must really be tuned to each other, and thus one cannot easily substitute newer, faster processors when they are developed.

[2] As we'll see later, it may be much more challenging than this implies. First of all, the computation will need some communication between the processors, which harms performance if it is not done carefully. Second, if one really wants a good speedup, one may need to take into account the fact that some squares require more computation work than others. More on this below.

[3] If one really wants a good NOW, though, one would need to have a faster network than CSIF's.

mail, file sharing, Internet connections and so on. But they are sometimes also used for parallel processing, with the network being used to pass messages between various machines which are cooperating on the same task; a NOW is thus an example of **message-passing hardware**.

Remember, the reason for buying a parallel processing machine is *speed*. You need to have a program that runs as fast as possible. That means that in order to write good parallel processing software, you must have a good knowledge of the underlying hardware. You must find also think of clever tricks for **load balancing,** i.e. keeping all the processors busy as much as possible. In the graphics ray-tracing application, for instance, suppose a ray is coming from the "northeast" section of the image, and is reflected by a solid object. Then the ray won't reach some of the "southwest" portions of the image, which then means that the processors assigned to those portions will not have any work to do which is associated with this ray. What we need to do is then try to give these processors some other work to do; the more they are idle, the slower our system will be.

2 Programming Paradigms

There are two main paradigms today in parallel-processing, **shared memory** and **message passing**. These distinctions can occur at either the software or hardware level. In other words, both software and hardware can be designed around both the shared-memory and message-passing paradigms. Thus for example, the UCD Registrar could run message-passing software such as the MPI package on their shared-memory hardware, while we could use the shared-memory software package Treadmarks on the message-passing NOW in CSIF.

2.1 World Views

To explain the two paradigms, we will use the term **nodes**, where roughly speaking one node corresponds to one processor, and use the following example:

> Suppose we wish to multiply an nx1 vector X by an nxn matrix A, putting the product in an nx1 vector Y, and we have p processors to share the work.

2.1.1 Shared-Memory

In the shared-memory paradigm, the arrays for A, X and Y would be held in common by all nodes. If for instance node 2 were to execute

```
Y[3] = 12;
```

and then node 15 were to subsequently execute

```
print("%d\n",Y[3]);
```

then the outputted value from the latter would be 12.

2.1.2 Message Passing

By contrast, in the message-passing paradigm, all nodes would have separate copies of A, X and Y (or maybe some nodes would not even have some or all of these arrays). In this case, in our example above, in order for node 2 to send this new value of Y[3] to node 15, it would have to execute some special function, which would be something like

```
send(15,12,"Y[3]");
```

and node 15 would have to execute some kind of **receive()** function.

The conventional wisdom is that the shared-memory paradigm is much easier to program in than the message-passing paradigm. The latter, however, may be easier to implement, and in some settings may have greater speed.

2.1.3 SIMD

Another paradigm is **Single Instruction, Multiple Data** (SIMD). This is almost entirely a hardware issue. You can think of it as a processor with a very large number of ALUs, except that they do not operate independently. Whenever the CPU issues an instruction, that same instruction is executed in lockstep by all the ALUs. Another difference is that each ALU has its own data registers.

In classical SIMD machines, the ALUs are arranged in a two-dimensional array. A typical instruction might be, say, SENDLEFT, which would mean sending the value in the ALU's register to the ALU on one's left (or nowhere, if the ALU is on the left edge).

This is common in image-processing applications, for example. Say we hae one ALU per pixel and we wish to replace each pixel value by the average of its neighboring values. The code might look like this:

```
set sum to 0
add left neighbor to sum
add right neighbor to sum
add top neighbor to sum
add bottom neighbor to sum
divide sum by 4
```

Again, remember that this is occurring simultaneously at all the ALUs, i.e. at all the pixels.

Some of the "MMX"-style chips operate under the SIMD paradigm.

2.2 Shared-Memory Example

Today, programming on shared-memory multiprocessors is typically done via **threading**. A **thread** is similar to a **process** in an operating system (OS), but with much less overhead. Threaded applications have become quite popular in even uniprocessor systems, and Unix, Windows, Python, Java and Perl all support threaded programming. One of the most famous threads packages is Pthreads.

2.2.1 How Threads Work on Multiprocessor Systems

Even if you have had some exposure to threaded programming before, it's important to understand how things change when we use it in a multiprocessor environment. To this end, let's first review how an OS schedules processes on a uniprocessor system.

Say persons X and Y are both running programs on the same uniprocessor machine. Since there is only one CPU, only one program is running at any given time, but they do "take turns." X's program will run for a certain amount of time, which we'll assume for concreteness is 50 milliseconds. After 50 milliseconds, a hardware timer will issue an interrupt, which will cause X's program to suspend and the OS to resume execution. The **state** of X's program at the time of the interrupt, i.e. the values in the registers etc., will be saved by the OS, and the OS will then restore the state of Y's program which had been saved at its last turn. Finally, the OS will execute a interrupt-return instruction, which will cause Y's program to restore execution in exactly the setting which it had at the end of its last turn. Note also that if the program which is currently running makes a **system call**, i.e. calls a function in the OS for input/output or other services, the program's turn ends before 50 ms.

But again, at any given time only one of the three programs (X, Y and the OS) is running. By contrast, on a multiprocessor system with k CPUs, at any given time k programs are running. When a turn for a program ends on a given CPU, again an interrupt occurs and the OS resumes execution, at which time it looks for another program to run.

Though we have been speaking in terms of programs, the proper term is **processes**. Say for instance that three people are running the GCC compiler right now on a certain machine. That would be only one program but three processes.

For the type of threads we are discussing here—**nonpreemptive** and system level—a thread essentially is a process. If for instance a program creates four threads, then all four will show up when one runs the **ps** command on a Unix system. The difference is that threads have much less overhead than do ordinary processes.

Threaded programming is natural for shared-memory multiprocessors, since it does share memory.[4] Just like the process pool is shared by all the processors, so is the thread pool. Whenever a processor finishes a timeslice for a thread, it goes to the thread pool to find another one to process. In that manner, there usually will be many threads executing truly simultaneously, i.e. we get real parallelism.

[4] On Unix systems, one can arrange for shared memory between processes by using **shmem()**. However, it has poor peformance and is unwieldy. It is much easier, more direct and more natural with threads.

2.2.2 Example

Here is an example of Pthreads programming:

```
1  // PrimesThreads.c
2
3  // threads-based program to find the number of primes between 2 and n;
4  // uses the Sieve of Eratosthenes, deleting all multiples of 2, all
5  // multiples of 3, all multiples of 5, etc.
6
7  // for illustration purposes only; NOT claimed to be efficient
8
9  // Unix compilation:  gcc -g -o primesthreads PrimesThreads.c -lpthread -lm
10
11 // usage:  primesthreads n
12
13 #include <stdio.h>
14 #include <math.h>
15 #include <pthread.h>  // required for threads usage
16
17 #define MAX_N 100000000
18 #define MAX_THREADS 25
19
20 // shared variables
21 int nthreads,  // number of threads (not counting main())
22     n,  // range to check for primeness
23     prime[MAX_N+1],  // in the end, prime[i] = 1 if i prime, else 0
24     nextbase;  // next sieve multiplier to be used
25 // lock for the shared variable nextbase
26 pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
27 // ID structs for the threads
28 pthread_t id[MAX_THREADS];
29
30 // "crosses out" all odd multiples of k
31 void crossout(int k)
32 {  int i;
33    for (i = 3; i*k <= n; i += 2)  {
34       prime[i*k] = 0;
35    }
36 }
37
38 // each thread runs this routine
39 void *worker(int tn)  // tn is the thread number (0,1,...)
40 {  int lim,base,
41        work = 0;  // amount of work done by this thread
42    // no need to check multipliers bigger than sqrt(n)
43    lim = sqrt(n);
44    do  {
45       // get next sieve multiplier, avoiding duplication across threads
46       // lock the lock
47       pthread_mutex_lock(&nextbaselock);
48       base = nextbase;
49       nextbase += 2;
50       // unlock
```

```
51        pthread_mutex_unlock(&nextbaselock);
52        if (base <= lim)  {
53           work++;  // log work done by this thread
54           // don't bother crossing out if base known composite
55           if (prime[base]) crossout(base);
56        }
57        else return work;
58     } while (1);
59  }
60
61  main(int argc, char **argv)
62  {  int nprimes,  // number of primes found
63         i,work;
64     n = atoi(argv[1]);
65     nthreads = atoi(argv[2]);
66     // mark all even numbers nonprime, and the rest "prime until
67     // shown otherwise"
68     for (i = 3; i <= n; i++)  {
69        if (i%2 == 0) prime[i] = 0;
70        else prime[i] = 1;
71     }
72     nextbase = 3;
73     // get threads started
74     for (i = 0; i < nthreads; i++)  {
75        // this call says to create a thread, record its ID in the array
76        // id, and get the thread started executing the function worker(),
77        // passing the argument i to that function
78        pthread_create(&id[i],NULL,worker,i);
79     }
80
81     // barrier, to wait for all done
82     for (i = 0; i < nthreads; i++)  {
83        // this call said to wait until thread number id[i] finishes
84        // execution, and to assign the return value of that thread to our
85        // local variable work here
86        pthread_join(id[i],&work);
87        printf("%d values of base done\n",work);
88     }
89
90     // report results
91     nprimes = 1;
92     for (i = 3; i <= n; i++)
93        if (prime[i])  {
94           nprimes++;
95        }
96     printf("the number of primes found was %d\n",nprimes);
97
98  }
```

To make our discussion concrete, suppose we are running this program with two threads. Suppose also the both threads are simultaneously most of the time. This will occur if they aren't competing for turns with other big threads. That in turn will occur if there are no other big threads, or more generally if the number of other big threads is less than or equal to the number of processors minus two.

Note the global variables:

```
int nthreads,  // number of threads (not counting main())
    n,  // range to check for primeness
    prime[MAX_N+1],  // in the end, prime[i] = 1 if i prime, else 0
    nextbase;  // next sieve multiplier to be used
pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
pthread_t id[MAX_THREADS];
```

This will require some adjustment for those who've been taught that global variables are "evil." All communication between processors in shared-memory systems[5] is via global variables, so if they are evil, they are a necessary evil. Personally I think there is nothing wrong with global variables anyway. See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.

As mentioned earlier, these are shared by all processors.[6] If one processor, for instance, assigns the value 0 to **prime[35]** in the function **crossout()**, then that variable will have the value 0 when accessed by any of the other processors as well. On the other hand, local variables have different values at each processor; for instance, the variable **i** in that function has a different value at each processor.

In the code

```
pthread_mutex_lock(&nextbaselock);
base = nextbase
nextbase += 2
pthread_mutex_unlock(&nextbaselock);
```

we see a **critical section** operation which is typical in shared-memory programming. In this context here, it means that we cannot allow more than one thread to execute

```
base = nextbase;
nextbase += 2;
```

at the same time. The calls to **pthread_mutex_lock()** and **pthread_mutex_unlock()** ensure this. If thread A is currently executing inside the critical section and thread B tries to lock the lock by calling **pthread_mutex_lock()**, the call will block until thread B executes **pthread_mutex_unlock()**.

Here is why this is so important: Say currently **nextbase** has the value 11. What we want to happen is that the next thread to read **nextbase** will then "cross out" all multiples of 11. But if we allow two threads to execute the critical section at the same time, the following may occur:

- thread A reads **nextbase**, setting its value of **base** to 11
- thread B reads **nextbase**, setting its value of **base** to 11

[5] And for that matter, within threaded programs on uniprocessor systems.

[6] Technically, we should say "shared by all threads" here, as a given thread does not always execute on the same processor, but at any instant in time each executing thread is at some processor, so the statement is all right.

- thread A adds 2 to **nextbase**, so that **nextbase** becomes 13
- thread B adds 2 to **nextbase**, so that **nextbase** becomes 15

Two problems would then occur:

- Both threads would do "crossing out" of multiples of 7, thus duplicating and thus a slowing down execution speed.
- We will never "cross out" multiples of 13.

Thus the lock is crucial to the correct (and speedy) execution of the program.

Note the **barrier**:

```
for (i = 0; i < nthreads; i++)  {
   pthread_join(id[i],&work);
   printf("%d values of base done\n",work);
}
```

Ignore the **printf()** call; the main purpose of this loop is to wait for all threads to finish. This is needed in order to prevent premature execution of the later code

```
for (i = 3; i <= n; i++)
   if (prime[i])  {
      nprimes++;
   }
```

resulting in possibly wrong output if we start counting primes before some threads are done.

Actually, barriers are more general than this. A barrier is simply a point in the code at which we must wait for all threads to reach before continuing. The threads do not necessarily have to exit at that point, as they do here. This is a very common operation in shared-memory programming, which we will return to later.

2.3 Message-Passing Example

Here we use the MPI system, a popular public-domain set of interface functions, callable from C/C++, to do message passing. We are again counting primes, though in this case using a **pipelining** method. It is similar to hardware pipelines, but in this case it is done in software, and each "stage" in the pipe is a different computer.

The program is self-documenting, via the comments.

```
/* this include file is mandatory */
#include <mpi.h>

/* MPI sample program; NOT INTENDED TO BE EFFICIENT as a prime
   finder, either in algorithm or implementation

   MPI (Message Passing Interface) is a popular package using
   the "message passing" paradigm for communicating between
   processors in parallel applications; as the name implies,
   processors communicate by passing messages using "send" and
   "receive" functions

   finds and reports the number of primes less than or equal to N

   uses a pipeline approach:  node 0 looks at all the odd numbers
   (i.e. has already done filtering out of multiples of 2) and
   filters out those that are multiples of 3, passing the rest
   to node 1; node 1 filters out the multiples of 5, passing
   the rest to node 2; in this simple example, we just have node
   2 filter out all the rest and then report the number of primes

   note that we should NOT have a node run through all numbers
   before passing them on to the next node, since we would then
   have no parallelism at all; on the other hand, passing on just
   one number at a time isn't efficient either, due to the high
   overhead of sending a message if it is a network (tens of
   microseconds until the first bit reaches the wire, due to
   software delay); thus efficiency would be greatly improved if
   each node saved up a chunk of numbers before passing them to
   the next node */

#define MAX_N 100000
#define PIPE_MSG 0  /* type of message containing a number to
                       be checked */
#define END_MSG 1  /* type of message indicating no more data will
                      be coming */

int NNodes,  /* number of nodes in computation*/
    N,  /* find all primes from 2 to N */
    Me,  /* my node number */
    ToCheck;  /* current number to check for passing on to next node;
                 stylistically this might be nicer as a local in
                 Node*(), but I have placed it here to dramatize
                 the fact that the globals are NOT shared among
                 the nodes */

double T1,T2;  /* start and finish times */

Init(Argc,Argv)
   int Argc;  char **Argv;

{  int DebugWait;

   N = atoi(Argv[1]);
   DebugWait = atoi(Argv[2]);
```

```
56
57    /* this loop is here to synchronize all nodes for debugging;
58       if DebugWait is specified as 1 on the command line, all nodes
59       wait here until the debugging programmer starts GDB at all
60       nodes and within GDB sets DebugWait to 0 to then proceed */
61    while (DebugWait) ;
62
63    /* mandatory to begin any MPI program */
64    MPI_Init(&Argc,&Argv);
65
66    /* puts the number of nodes in NNodes */
67    MPI_Comm_size(MPI_COMM_WORLD,&NNodes);
68    /* puts the node number of this node in Me */
69    MPI_Comm_rank(MPI_COMM_WORLD,&Me);
70
71    /* OK, get started; first record current time in T1 */
72    if (Me == 2) T1 = MPI_Wtime();
73 }
74
75 Node0()
76
77 {  int I,Dummy,
78        Error;  /* not checked in this example */
79    for (I = 1; I <= N/2; I++)  {
80       ToCheck = 2 * I + 1;
81       if (ToCheck > N) break;
82       /* MPI_Send  --  send a message
83                parameters:
84                pointer to place where message is to be drawn from
85                number of items in message
86                item type
87                destination node
88                message type ("tag") programmer-defined
89                node group number (in this case all nodes) */
90       if (ToCheck % 3 > 0)
91          Error = MPI_Send(&ToCheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
92    }
93    Error = MPI_Send(&Dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
94 }
95
96 Node1()
97
98 {  int Error,  /* not checked in this example */
99        Dummy;
100   MPI_Status Status;  /* see below */
101
102   while (1)  {
103      /* MPI_Recv  --  receive a message
104               parameters:
105               pointer to place to store message
106               number of items in message (see notes on
107                  this at the end of this file)
108               item type
109               accept message from which node(s)
110               message type ("tag"), programmer-defined (in this
```

```
111           case any type)
112                node group number (in this case all nodes)
113                status (see notes on this at the end of this file) */
114       Error = MPI_Recv(&ToCheck,1,MPI_INT,0,MPI_ANY_TAG,
115                   MPI_COMM_WORLD,&Status);
116       if (Status.MPI_TAG == END_MSG) break;
117       if (ToCheck % 5 > 0)
118          Error = MPI_Send(&ToCheck,1,MPI_INT,2,PIPE_MSG,MPI_COMM_WORLD);
119    }
120    /* now send our end-of-data signal, which is conveyed in the
121       message type, not the message (we have a dummy message just
122       as a placeholder */
123    Error = MPI_Send(&Dummy,1,MPI_INT,2,END_MSG,MPI_COMM_WORLD);
124 }
125
126 Node2()
127
128 {  int ToCheck,  /* current number to check from Node 0 */
129        Error,  /* not checked in this example */
130        PrimeCount,I,IsComposite;
131    MPI_Status Status;  /* see below */
132
133    PrimeCount = 3;  /* must account for the primes 2, 3 and 5, which
134                        won't be detected below */
135    while (1)  {
136       Error = MPI_Recv(&ToCheck,1,MPI_INT,1,MPI_ANY_TAG,
137                   MPI_COMM_WORLD,&Status);
138       if (Status.MPI_TAG == END_MSG) break;
139       IsComposite = 0;
140       for (I = 7; I*I <= ToCheck; I += 2)
141          if (ToCheck % I == 0)  {
142     IsComposite = 1;
143     break;
144  }
145       if (!IsComposite) PrimeCount++;
146    }
147    /* check the time again, and subtract to find run time */
148    T2 = MPI_Wtime();
149    printf("elapsed time = %f\n",(float)(T2-T1));
150    /* print results */
151    printf("number of primes = %d\n",PrimeCount);
152 }
153
154 main(argc,argv)
155    int argc; char **argv;
156
157 {  Init(argc,argv);
158    /* note:  instead of having a switch statement, we could write
159       three different programs, each running on a different node */
160    switch (Me)  {
161       case 0:  Node0();
162                break;
163       case 1:  Node1();
164                break;
165       case 2:  Node2();
```

```
166     };
167     /* mandatory for all MPI programs */
168     MPI_Finalize();
169  }
170
171  /* explanation of "number of items" and "status" arguments at the end
172     of MPI_Recv():
173
174     when receiving a message you must anticipate the longest possible
175     message, but the actual received message may be much shorter than
176     this; you can call the MPI_Get_count() function on the status
177     argument to find out how many items were actually received
178
179     the status argument will be a pointer to a struct, containing the
180     node number, message type and error status of the received
181     message
182
183     say our last parameter is Status; then Status.MPI_SOURCE
184     will contain the number of the sending node, and
185     Status.MPI_TAG will contain the message type; these are
186     important if used MPI_ANY_SOURCE or MPI_ANY_TAG in our
187     node or tag fields but still have to know who sent the
188     message or what kind it is */
```

3 Message-Passing Mechanisms

3.1 Message-Passing Hardware

3.1.1 Hypercubes

A popular class of parallel machines used to be that of **hypercubes**. Intel sold them, for example. A hypercube would consist of some number of ordinary Intel processors, with each processor having some memory and serial I/O hardward for connection to its "neighbor" processors.

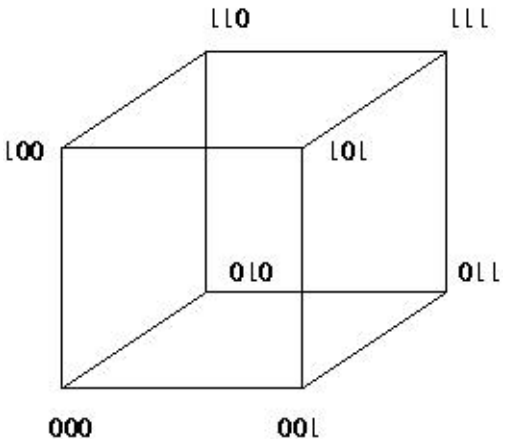
Hypercubes proved to be too expensive for the type of performance they could achieve, and the market was small anyway. Thus they are not common today, but they are still important, in that the algorithms developed for them have become quite popular for use on general machines. In this section we will discuss architecture, algorithms and software for such machines.

Definitions A **hypercube** of dimension d consists of $D = 2^d$ **processing elements** (PEs), i.e. processor-memory pairs, We refer to such a cube as a **d-cube**.

The PEs in a d-cube will have numbers 0 through D-1. Let $(c_{d-1}, ..., c_0)$ be the base-2 representation of a PE's number. The PE has fast point-to-point links to d other PEs, which we will call its **neighbors**. Its i^{th} neighbor has number $(c_{d-1}, ..., 1 - c_{i-1}, ..., c_0)$.[7]

[7] Note that we number the digits from right to left, with the rightmost digit being digit 0.

For example, consider a hypercube having D = 16, i.e. d = 4. The PE numbered 1011, for instance, would have four neighbors, 0011, 1111, 1001 and 1010.

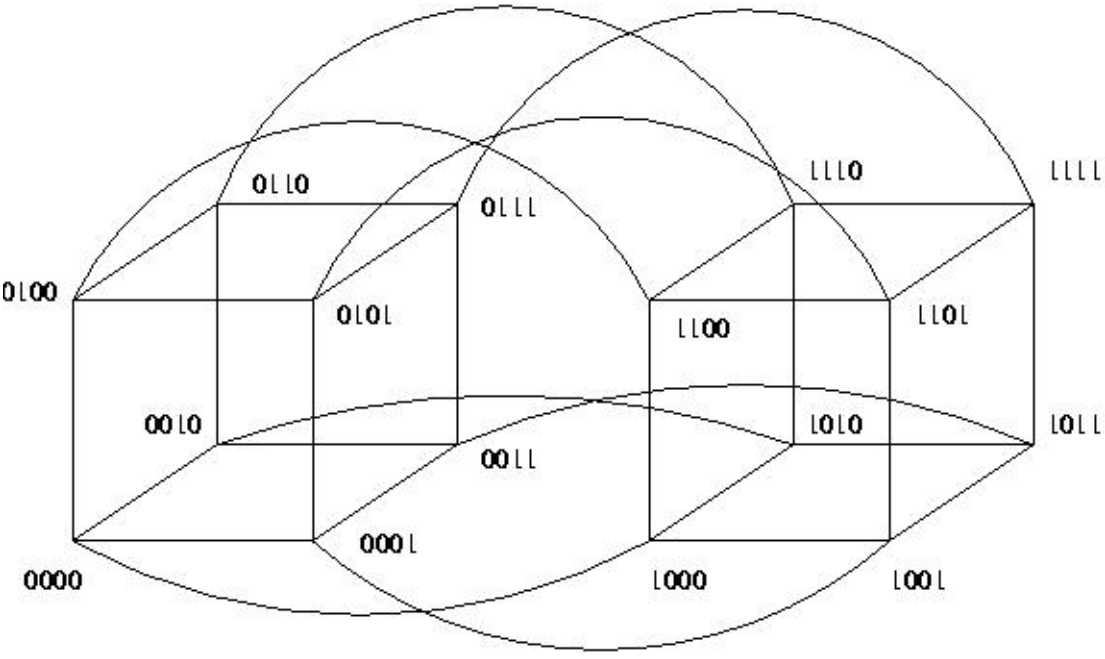


It is sometimes helpful to build up a cube from the lower-dimensional cases. To build a (d+1)-dimensional cube from two d-dimensional cubes, just follow this recipe:

(a) Take a d-dimensional cube and duplicate it. Call these two cubes subcube 0 and subcube 1.

(b) For each pair of same-numbered PEs in the two subcubes, add a binary digit 0 to the front of the number for the PE in subcube 0, and add a 1 in the case of subcube 1. Add a link between them.

The following figure shows how a 4-cube can be constructed in this way from two 3-cubes:



Given a PE of number $(c_{d-1}, ..., c_0)$ in a d-cube, we will discuss the i-cube to which this PE belongs, meaning all PEs whose first d-i digits match this PE's.[8] Of all these PEs, the one whose last i digits are all 0s is called the **root** of this i-cube.

For the 4-cube and PE 1011 mentioned above, for instance, the 2-cube to which that PE belongs consists of 1000, 1001, 1010 and 1011—i.e. all PEs whose first two digits are 10—and the root is 1000.

Given a PE, we can split the i-cube to which it belongs into two (i-1)-subcubes, one consisting of those PEs whose digit i-1 is 0 (to be called subcube 0), and the other consisting of those PEs whose digit i-1 is 1 (to be called subcube 1). Each given PE in subcube 0 has as its **partner** the PE in subcube 1 whose digits match those of the given PE, except for digit i-1.

To illustrate this, again consider the 4-cube and the PE 1011. As an example, let us look at how the 3-cube it belongs to will split into two 2-cubes. The 3-cube to which 1011 belongs consists of 1000, 1001, 1010, 1011, 1100, 1101, 1110 and 1111. This 3-cube can be split into two 2-cubes, one being 1000, 1001, 1010 and 1011, and the other being 1100, 1101, 1110 and 1111. Then PE 1000 is partners with PE 1100, PE 1001 is partners with PE 1101, and so on.

Each link between two PEs is a dedicated connection, much preferable to the shared link we have when we run, say, MPI, on a collection of workstations on an Ethernet. On the other hand, if one PE needs to communicate with a non-neighbor PE, multiple links (as many as d of them) will need to be traversed. Thus the nature of the communications costs here is much different than for a network of workstations, and this must be borne in mind when developing programs.

3.1.2 Networks of Workstations (NOWs)

As of this writing (early 2004), the commercial market for shared-memory hardware is strong, with the machines being used for heavy-duty server applications, such as for large databases and World Wide Web sites. The conventional wisdom is that these applications require the efficiency that good shared-memory hardware can provide. Nevertheless, the prices of these systems are enormous, and it is very unclear as to whether they can remain commercially viable.

Instead, the most promising alternative today appears to be networks of workstations (NOWs). It is much cheaper to purchase a set of workstations[9] and network them for use as parallel processing systems. They are of course individual machines, capable of the usual uniprocessor, nonparallel applications, but by networking them together and using parallel-processing software environments, we can form very powerful parallel systems.

The networking does result in a significant loss of performance. This will be discussed later. But even without these techniques, the price/performance ratio in NOW is much superior in many applications to that of specialized shared-memory or hypercube hardware.

One factor which can be key to the success of a NOW is to use a fast network, both in terms of hardware and network protocol. Ordinary Ethernet and TCP/IP are fine for the applications envisioned by the original

[8] Note that this is indeed an i-dimensional cube, because the last i digits are free to vary.

[9] I am including PCs in this category, as they are certainly as powerful today as the machines traditionally called "workstations."

designers of the Internet, e.g. e-mail and file transfer, but is slow in the NOW context. A good network for a NOW is, for instance, Infiniband. For information on network protocols, e.g. for example `www.rdmaconsortium.org`.

3.1.3 Hardware Issues

Multiprocessor architects use the terms **latency** and **bandwidth** to provide a rough description of the performance potential of an interconnect. Latency means time it takes for a bit to travel from source to destination. This includes queuing delays within network nodes, software delays (as a bit travels through software layers, e.g. from application program to OS to I/O hardware), etc. Bandwidth means the number of bits which can be sent simultaneously.

In a message-passing system on an Ethernet, node-to-node communication latency is quite high, as high as hundreds of microseconds, which is huge compared to memory-latency time for (at least the smaller) shared-memory systems. This can be greatly reduced by using a special network, but in any case, when writing application software, failure to write the code in such a way as to deal with this latency issue can severely reduce program speed, sometimes even to the point at which running on a single machine is faster than running in parallel.

One remedy that one can try is to use **nonblocking** versions of Send() functions to reduce the effect of this latency. This means that a node does not have to wait for a message transmission to occur, but can instead work on other subtasks in the interim.

The newest message-passing hardware/software systems implement **active messages**. Here latency is actually reduced, because a message will be deposited directly into the application program's buffer at the receiver, rather than being buffered by the operating system first and then copied a second time to the application program's address space.

For example, consider the code

```
int X[1000000];
...
Recv(X);
```

The array X will be sent to this node by some other node, with a call something like

```
Send(X);
```

As this message arrives at the receiving node via the Ethernet, the Ethernet card at that node will cause a CPU interrupt. The interrupt service routine, i.e. the OS, will copy the incoming data from ports in the Ethernet card to a buffer in the OS. When done, that data will then be sent by the OS to the application program, which will, via Recv(), copy the data to X. In other words, we have only one copy operation to memory, not two, which can save a lot of time.

Another way to deal with latency is to hide it, rather than reduce it, using threading. If we are using message-passing, for instance, instead of waiting a long time to receive a message, we can switch to a different thread and do other useful work for our parallel processing task while we are waiting.

3.1.4 Message Passing on Shared-Memory Machines

Even if one has shared-memory hardware, one can still do message-passing on it, say with MPI. In fact, IBM even produces its own version of MPI, especially tailored to IBM's shared-memory machines.

Why would one do this? Well, even though most people in the parallel processing community believe that the shared-memory programming paradigm is clearer than the message-passing paradigm, there is general agreement that if one wants to really get as much speed as possible on a given platform, message-passing is better able to attain top speed than shared-memory programming. Another reason for using message passing on a shared-memory machine, using MPI, is that the code is then usable on (if not optimized for) use on almost any platform.

3.2 Message-Passing Software

The first widely-used message-passing software was PVM (Parallel Virtual Machine); see the links at `http://heather.cs.ucdavis.edu/~matloff/pvm.html`. It was developed at Oak Ridge National Laboratory. It is still in wide use today, but its successor MPI (Message Passing Interface); see the links at `http://heather.cs.ucdavis.edu/~matloff/mpi.html`. It was developed at Argonne National Laboratory is probably the more popular package now. Both PVM and MPI are public-domain.

4 Shared-Memory Mechanisms

4.1 Hardware Issues

As mentioned in the message-passing setting above, when writing application software to run on such systems, failure to write the code in such a way as to deal with hardware issues can severely reduce program speed, sometimes even to the point at which running on a single machine is faster than running in parallel.

4.2 Shared-Memory through Hardware

(Note: In spite of the word *hardware* in the title of this section, you will see later that many of the issues discussed here will also arise in software contexts.)

The term **shared memory** means that the processors all share a common address space. Say this is occurring at the hardware level, and we are using Intel Pentium CPUs. Suppose processor P3 issues the instruction

```
movl 200, %eabx
```

which reads memory location 200 and places the result in the EAX register in the CPU. If processor P4 does the same, they both will be referring to the same physical memory cell. In non-shared-memory machines, each processor has its own private memory, and each one will then have its own location 200, completely independent of the locations 200 at the other processors' memories.

Say a program contains a global variable X and a local variable Y on share-memory hardware (and we use shared-memory software). If for example the compiler assigns location 200 to the variable X, i.e. `&X = 200`, then the point is that all of the processors will have that variable in common, because any processor which issues a memory operation on location 200 will access the same physical memory cell.

On the other hand, each processor will have its own separate run-time stack,[10] and thus each processor will have its own independent copy of the local variable Y.

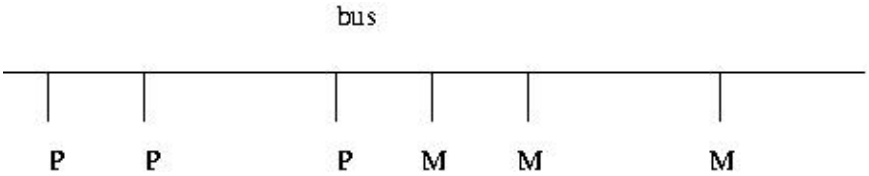
To make the meaning of "shared memory" more concrete, suppose we have a bus-based system, with all the processors and memory attached to the bus. Let us compare the above variables X and Y here. Suppose again that the compiler assigns X to memory location 200. Then in the machine language code for the program, every reference to X will be there as 200. Every time an instruction involving X is executed by a CPU, that CPU will put 200 into its Memory Address Register (MAR), from which the 200 flows out on the address lines in the bus, and goes to memory. This will happen in the same way no matter which CPU it is. Thus the same physical memory location will end up being accessed, no matter which CPU generated the reference.

By contrast, say the compiler assigns Y to something like ESP+8, the third item on the stack.[11] Each CPU will have its own current value for ESP, so the stacks of the various CPUs will be separate.[12]

4.2.1 Placement of Memory Modules

The placement of the memory modules is quite important.

Symmetric Multiprocessor Let's first consider the following structure:



Here and below:

[10] Still in shared memory, but a separate stack for each processor, since each CPU has a different value in its SP register.

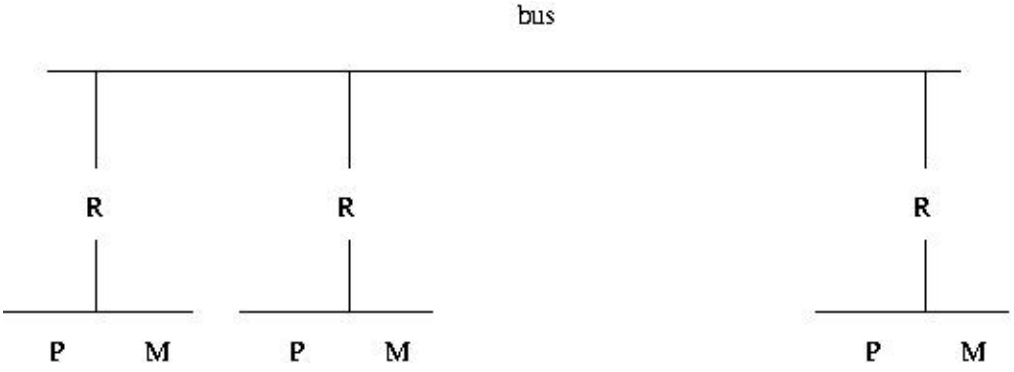
[11] ESP is the number of the stack pointer register in the Pentium.

[12] Again, the stacks will be in the physical shared memory, and thus P3, say, could theoretically access P8's stack, say if there were an erroneous pointer value. But even that would not occur if we are using virtual memory and thus have protections against this.

- The Ps are processors, e.g. off-the-shelf chips such as Pentiums.
- The Ms are **memory modules**. These are physically separate objects, e.g. separate boards of memory chips. It is typical that there will be the same number of Ms as Ps, but it does not have to be this way. In the shared-memory case, the Ms collectively form the entire shared address space, but with the addresses being assigned to the Ms in one of two ways:
  - (a)
    High-order interleaving. Here consecutive addresses are in the same M (except at boundaries). For example, suppose for simplicity that our memory consists of addresses 0 through 1023, and that there are four Ms. Then M0 would contain addresses 0-255, M1 would have 256-511, M2 would have 512-767, and M3 would have 768-1023.[13]
  - (b)
    Low-order interleaving. Here consecutive addresses are in consecutive M's (except when we get to the right end). In the example above, if we used low-order interleaving, then address 0 would be in M0, 1 would be in M1, 2 would be in M2, 3 would be in M3, 4 would be back in M0, 5 in M1, and so on.
- To make sure only one P uses the bus at a time, standard bus arbitration signals and/or arbitration devices are used.
- There may also be **coherent caches**, which we will discuss later.

The shared-memory, bus-connected case shown here is called a **symmetric multiprocessor** (SMP). It is also referred to as a Uniform Memory Access (UMA) structure, meaning that all CPUs have the same access time to memory. Except for the negligible difference in bus propagation delays, each processor takes the same time to acquire the bus and then access memory.

A NUMA Example By contrast, look at this version:



Each P/M/R set here is called a **processing element** (PE). Note that each PE has its own local bus, and is also connected to the global bus via R, the router.

[13] Miscellaneous gates ("glue") would be used so that the correct M would recognize a bus address as being for that M.

Suppose for example that P3 needs to access location 200, and suppose that high-order interleaving is used.[14] If location 200 is in M3, then P3's request is satisfied by the local bus.[15] On the other hand, suppose location 200 is in M8. Then the R3 will notice this, and put the request on the global bus, where it will be seen by R8, which will then copy the request to the local bus at PE8, where the request will be satisfied. (E.g. if it was a read request, then the response will go back from M8 to R8 to the global bus to R3 to P3.)

It should be obvious now where NUMA gets its name. P8 will have much faster access to M8 than P3 will to M8, if none of the buses is currently in use—and if say the global bus is currently in use, P3 will have to wait a long time to get what it wants from M8.

These days NUMA systems are really in vogue. One of the attractive features of NUMA is that by good programming we can exploit the nonuniformity. In matrix problems, for example, we can write our program so that, for example, P8 usually works on those rows of the matrix which are stored in M8, P3 usually works on those rows of the matrix which are stored in M3, etc. In order to do this, we need to make use of the C language's & address operator, and have some knowledge of the memory hardware structure, i.e. the interleaving.

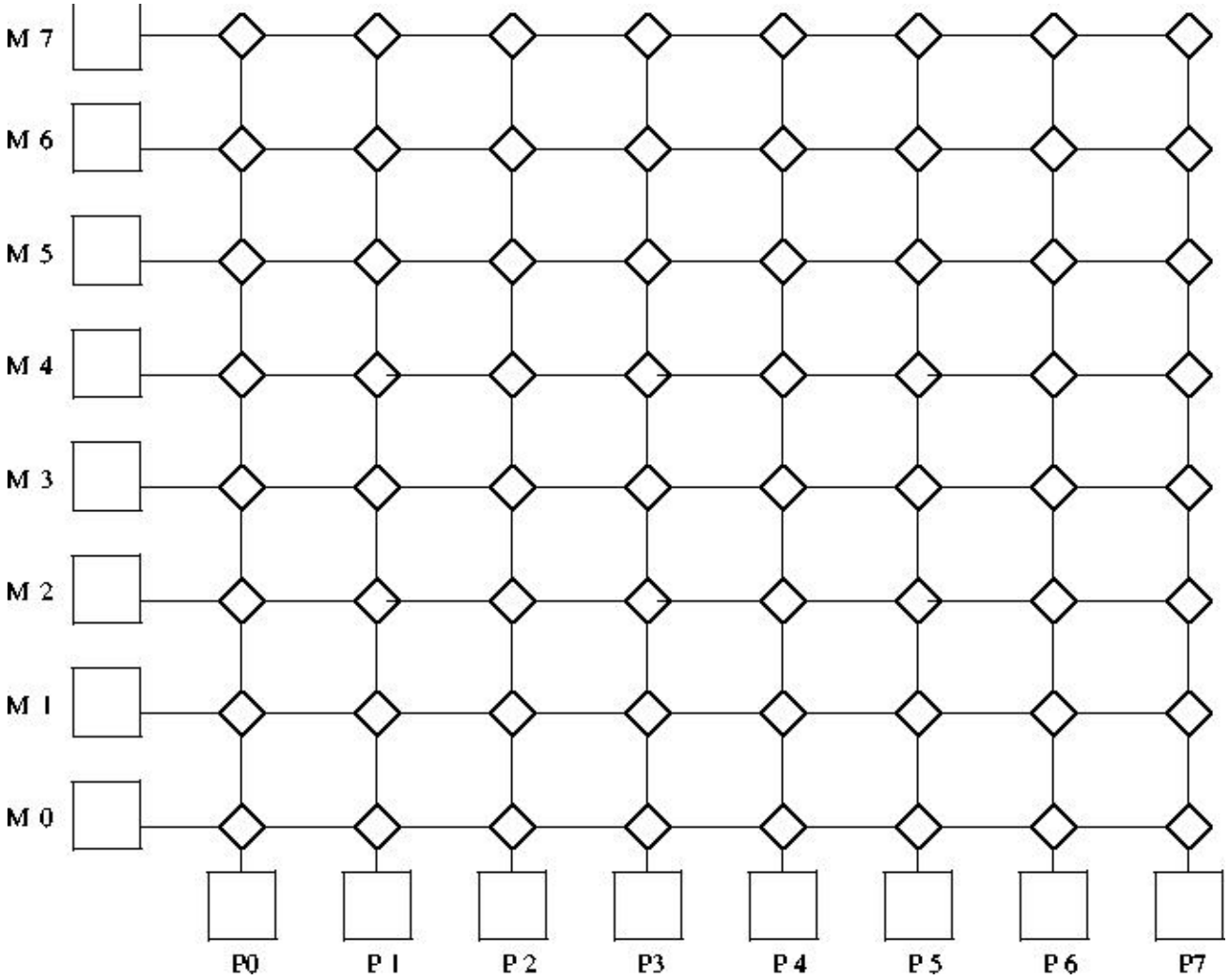
4.2.2 Interconnect Topologies

The problem with a bus connection, of course, is that there is only one pathway for communication, and thus only one processor can access memory at the same time. If one has more than, say, two dozen processors are on the bus, the bus becomes saturated, even if traffic-reducing methods such as adding caches are used. Thus multipathway topologies are used for all but the smallest systems. In this section we look at two alternatives to a bus topology.

Crossbar Interconnects Consider a UMA shared-memory system with n processors and n memory modules. The a crossbar connection would provide n^2 pathways. E.g. for n = 8:

[14] Low-order interleaving would probably be disastrous here.

[15] This sounds similar to the concept of a cache. However, it is very different. A cache contains a local copy of some data stored elsewhere. Here it is the data itself, not a copy, which is being stored locally.



Generally serial communication is used from node to node, with a packet containing information on both source and destination address. E.g. if P2 wants to read from M5, the source and destination will be 3-bit strings in the packet, coded as 010 and 101, respectively. The packet will also contain bits which specify which word within the module we wish to access, and bits which specify whether we wish to do a read or a write. In the latter case, additional bits are used to specify the value to be written.

Each diamond-shaped node has two inputs (bottom and right) and two outputs (left and top), with buffers at the two inputs. If a buffer fills, there are two design options: (a) Have the node from which the input comes block at that output. (b) Have the node from which the input comes discard the packet, and retry later, possibly outputting some other packet for now. If the packets at the heads of the two buffers both need to go out the same output, the one (say) from the bottom input will be given priority.

In this UMA setting,[16] there would also be a return network of the same type, with this one being memory

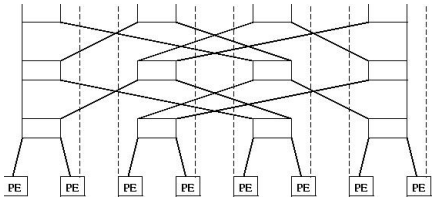
[16] Note that we are calling it UMA because processors and memory modules are not paired together. The access is not really very uniform.

→ processor, to return the result of the read requests.[17]

A NUMA version of this is also possible. It is not shown here, but the difference would be that at the bottom edge we would have the PEi and at the left edge the memory modules Mi would be replaced by lines which wrap back around to PEi.[18]

Crossbar switches are too expensive for large-scale systems, but are useful in some small systems. The 16-CPU Sun Microsystems Enterprise 10000 system includes a 16x16 crossbar.

Omega Interconnects These are multistage networks similar to crossbars, but with fewer paths. Here is an example of a NUMA 8x8 system:



Recall that each PE is a processor/memory pair. PE3, for instance, consists of P3 and M3.

Note the fact that at the third stage of the network (top of picture), the outputs are routed back to the PEs, each of which consists of a processor and a memory module.[19]

In a UMA version, we would see Pi instead of PEi at the bottom edge, and would see Mi at the top edge instead of dashed lines leading downward.

At each network node (the nodes are the three rows of rectangles), the output routing is done by destination bit. Let's number the stages here 0, 1 and 2, starting from the bottom stage, number the nodes within a stage 0, 1, 2 and 3 from left to right, number the PEs from 0 to 7, left to right, and number the bit positions in a destination address 0, 1 and 2, starting from the most significant bit. Then at stage i, bit i of the destination address is used to determine routing, with a 0 meaning routing out the left output, and 1 meaning the right one.

Say P2 wishes to read from M5. It sends a read-request packet, including 5 = 101 as its destination address, to the switch in stage 0, node 1. Since the first bit of 101 is 1, that means that this switch will route the packet out its right-hand output, sending it to the switch in stage 1, node 3. The latter switch will look at the next bit in 101, a 0, and thus route the packet out its left output, to the switch in stage 2, node 2. Finally, that switch will look at the last bit, a 1, and output out its right-hand output, sending it to PE5, as desired. M5 will process the read request, and send a packet back to PE2, along the same

Again, if two packets at a node want to go out the same output, one must get priority (let's say it is the one from the left input).

[17] For safety's sake, i.e. fault tolerance, even writes are typically acknowledged in multiprocessor systems.

[18] Similar to the Omega network shown below.

[19] The picture may be cut off somewhat at the top and left edges. The upper-right output of the rectangle in the top row, leftmost position should connect to the dashed line which leads down to the second PE from the left. Similarly, the upper-left output of that same rectangle is a dashed lined, possibly invisible in your picture, leading down to the leftmost PE.

Here is how the more general case of N = 2^n PEs works. Again number the rows of switches, and switches within a row, as above. So, S_{ij} will denote the switch in the i-th row from the bottom and j-th column from the left (starting our numbering with 0 in both cases). Row i will have a total of N input ports I_{ik} and N output ports O_{ik}, where k = 0 corresponds to the leftmost of the N in each case. Then if row i is not the last row ($i < n - 1$), O_{ik} will be connected to I_{jm}, where j = i+1 and

$$m = (2k + \lfloor (2k)/N \rfloor) \; mod \; N \tag{1}$$

If row i is the last row, then O_{ik} will be connected to, PE k.

Comparative Analysis In the world of parallel architectures, a key criterion for a proposed feature is **scalability**, meaning how well the feature performs as we go to larger and larger systems. Let n be the system size, either the number of processors and memory modules, or the number of PEs. Then we are interested in how fast the latency, bandwidth and cost grow with n:

criterion	bus	Omega	crossbar
latency	O(1)	$\mathrm{O(\log_2 n)}$	O(n)
bandwidth	O(1)	O(n)	O(n)
cost	O(1)	$\mathrm{O(n \log_2 n)}$	$\mathrm{O(n^2)}$

Let us see where these expressions come from, beginning with a bus: No matter how large n is, the time to get from, say, a processor to a memory module will be the same, thus O(1). Similarly, no matter how large n is, only one communication can occur at a time, thus again O(1).[20]

Again, we are interested only in "O( )" measures, because we are only interested in growth rates as the system size n grows. For instance, if the system size doubles, the cost of a crossbar will quadruple; the $O(n^2)$ cost measure tells us this, with any multiplicative constant being irrelevant.

For Omega networks, it is clear that $log_2 n$ network rows are needed, hence the latency value given. Also, each row will have n/2 switches, so the number of network nodes will be O(n $log_2 n$). This figure then gives the cost (in terms of switches, the main expense here). It also gives the bandwidth, since the maximum number of simultaneous transmissions will occur when all switches are sending at once.

Similar considerations hold for the crossbar case.

The crossbar's big advantage is that it is guaranteed that n packets can be sent simultaneously, providing they are to distinct destinations.[21]

That is <u>not</u> true for Omega-networks. If for example, PE0 wants to send to PE3, and at the same time PE4 wishes to sent to PE2, the two packets will clash at the leftmost node of stage 1, where the packet from PE0 will get priority.

[20] Note that the '1' in "O(1)" does not refer to the fact that only one communication can occur at a time. If we had, for example, a two-bus system, the bandwidth would still be O(1), since multiplicative constants do not matter. What O(1) means, again, is that as n grows, the bandwidth stays at a multiple of 1, i.e. stays constant.

[21] If two or more go to the same destination, they couldn't be satisfied simultaneously anyway, unless dual-port memory were used.

On the other hand, a crossbar is very expensive, and thus is dismissed out of hand in most modern systems. Note, though, that an equally troublesom aspect of crossbars is their high latency value; this is a big drawback when the system is not heavily loaded.

The bottom line is that Omega-networks amount to a compromise between buses and crossbars, and for this reason have become popular.

4.2.3 Test-and-Set

Consider a bus-based UMA system. In addition to whatever memory read and memory write instructions the processor included, say LD and ST, there would also be a TAS instruction.[22] This instruction would control a TAS pin on the processor chip, and the pin in turn would be connected to a TAS line on the bus.

Applied to a location L in memory and a register R, say, TAS does the following:

```
copy L to R
if R is 0 then write 1 to L
```

And most importantly, these operations are done in an **atomic** manner; no bus transactions by other processors may occur between the two steps.

The TAS operation is applied to variables used as **locks**. Let's say that 1 means locked and 0 unlocked. Then the guarding of a critical section C by a lock variable L would be done by having the following code in the program being run:

```
TRY:  TAS R,L
      JNZ TRY
  C:  ...   ; start of critical section
      ...
      ...   ; end of critical section
      MOV L,0  ; unlock
```

where of course JNZ is a jump-if-nonzero instruction, and we are assuming that the copying from the Memory Data Register to R results in the processor N and Z flags (condition codes) being affected.

A **critical section** is a portion of a program in which we cannot have more than one processor execute at a time. For instance, consider an airline reservation system. If a flight has only one seat left, we want to avoid giving it to two different customers who might be talking to two agents at the same time. The lines of code in which the seat is finally assigned (the **commit** phase, in database terminology) is then a critical section.

In crossbar or Ω-network systems, some 2-bit field in the packet must be devoted to transaction type, say 00 for Read, 01 for Write and 10 for TAS. In a sytem with 16 CPUs and 16 memory modules, say, the packet might consist of 4 bits for the CPU number, 4 bits for the memory module number, 2 bits for the transaction type, and 32 bits for the data (for a write, this is the data to be written, while for a read, it would be the requested value, on the trip back from the memory to the CPU).

[22] This discussion is for a mythical machine, but any real system works in this manner.

But note that the atomicity here is best done at the memory, i.e. some hardware should be added at the memory so that TAS can be done; otherwise, an entire processor-to-memory path (say in a UMA system) would have to be locked up for a fairly long time, obstructing even the packets which go to other memory modules.

There are many variations of test-and-set, so don't expect that all processors will have an instruction with this name, but they all will have some kind of synchronization instruction like it.

Note carefully that in many settings it may not be crucial to get the most up-to-date value of a variable. For example, a program may have a data structure showing work to be done. Some processors occasionally add work to the queue, and others take work from the queue. Suppose the queue is currently empty, and a processor adds a task to the queue, just as another processor is checking the queue for work. As will be seen later, it is possible that even though the first processor has written to the queue, the new value won't be visible to other processors for some time. But the point is that if the second processor does not see work in the queue (even though the first processor has put it there), the program will still work correctly, albeit with some performance loss.

4.2.4 Cache Coherency

Consider, for example, a bus-based system. Relying purely on TAS for interprocessor synchronization would be unthinkable: As each processor contending for a lock variable spins in the loop shown above, it is adding tremendously to bus traffic.

An answer is to have caches at each processor.[23] These will to store copies of the values of lock variables. (Of course, non-lock variables are stored too. However, the discussion here will focus on effects on lock variables.) The point is this: Why keep looking at a lock variable L again and again, using up the bus bandwidth? L may not change value for a while, so why not keep a copy in the cache, avoiding use of the bus?

The answer of course is that eventually L will change value, and this causes some delicate problems. Say for example that processor P5 wishes to enter a critical section guarded by L, and that processor P2 is already in there. During the time P2 is in the critical section, P5 will spin around, always getting the same value for L (1) from C5, P5's cache. When P2 leaves the critical section, P2 will set L to 0—and now C5's copy of L will be incorrect. This is the **cache coherency problem**, inconsistency between caches.

A number of solutions have been devised for this problem. For bus-based systems, **snoopy** protocols of various kinds are used, with the word "snoopy" referring to the fact that all the caches monitor ("snoop on") the bus, watching for transactions made by other caches.

The most common protocols are the **invalidate** and **update** types. This relation between these two is somewhat analogous to the relation between **write-back** and **write-through** protocols for caches in uniprocessor systems:

- Under an invalidate protocol, when a processor writes to a variable in a cache, it first (i.e. before

[23] The reader may wish to review the basics of caches. See for example http://heather.cs.ucdavis.edu/ matloff/50/PLN/CompOrganization.pdf.

actually doing the write) tells each other cache to mark as invalid its cache line (if any) which contains a copy of the variable.[24] Those caches will be updated only later, the next time their processors need to access this cache line.

- For an update protocol, the processor which writes to the variable tells all other caches to immediately update their cache lines containing copies of that variable with the new value.

Let's look at an outline of how one implementation (many variations exist) of an invalidate protocol would operate:

In the scenario outlined above, when P2 leaves the critical section, it will write the new value 0 to L. Under the invalidate protocol, P2 will post an invalidation message on the bus. All the other caches will notice, as they have been monitoring the bus. They then mark their cached copies of the line containing L as invalid.

Now, the next time P5 executes the TAS instruction—which will be very soon, since it is in the loop shown above—P5 will find that the copy of L in C5 is invalid. It will respond to this cache miss by going to the bus, and requesting P2 to supply the "real" (and valid) copy of the line containing L.

But there's more. Suppose that all this time P6 had also been executing the loop shown above. Then P5 and P6 may have to contend with each other; whoever manages to grab possession of the bus first[25] will be the one who ends up finding that L = 0. Let's say that that one is P6. P6 then executes the TAS again, finds L = 0, and then enters the critical section. An instant later, P5 P6 relinquishes the bus, P5 tries its execution of the TAS again. P5 acquires a valid copy of L now, but L will be 1 at this time, so P5 must resume executing the loop. P5 will then continue to use its valid local copy of L each time it does the TAS, until P6 leaves the critical section, writes 0 to L, and causes another cache miss at P5, etc.

At first the update approach seems obviously superior, and actually, if our shared, cacheable[26] variables were only lock variables, this might be true.

But consider a shared, cacheable vector. Suppose the vector fits into one block, and that we write to each vector element sequentially. Under an update policy, we would have to send a new message on the bus/network for each component, while under an invalidate policy, only one message (for the first component) would be needed. If during this time the other processors do not need to access this vector, all those update messages, and the bandwidth they use, would be wasted.

Or suppose for example we have code like

```
Sum += X[I];
```

in the middle of a **for** loop. Under an update protocol, we would have to write the value of Sum back many times, even though the other processors may only be interested in the final value when the loop ends. (This would be true, for instance, if the code above were part of a critical section.)

[24] we will follow commonly-used terminology here, distinguishing between a *cache line* and a *memory block*. Memory is divided in blocks, some of which have copies in the cache. The cells in the cache are called *cache lines*. So, at any given time, a given cache line is either empty or contains a copy (valid or not) of some memory block.

[25] Again, remember that ordinary bus arbitration methods would be used.

[26] Many modern processors, including Pentium and MIPS, allow the programmer to mark some blocks as being noncacheable.

If you have previously studied **write-back** and **write-through** cache policies on uniprocessor systems, you should note the similarities of the issues here. Note, though, that the problems are exacerbated here, due to the high cost of communication, e.g. bus contention.

Thus the invalidate protocol works well for some kinds of code, while update works better for others. The CPU designers must try to anticipate which protocol will work well across a broad mix of applications.[27]

Now, how is cache coherency handled in non-bus shared-memory systems, say crossbars? Here the problem is more complex. Think back to the bus case for a minute: The very feature which was the biggest negative feature of bus systems—the fact that there was only one path between components made bandwidth very limited—is a very positive feature in terms of cache coherency, because it makes broadcast very easy: Since everyone is attached to that single pathway, sending a message to all of them costs no more than sending it to just one—we get the others for free. That's no longer the case for multipath systems. In such systems, extra copies of the message must be created for each path, adding to overall traffic.

A solution is to send messages only to "interested parties." In **directory-based** protocols, a list is kept of all caches which currently have valid copies of all blocks. In one common implementation, for example, while P2 is in the critical section above, it would be the **owner** of the block containing L. (Whoever is the latest node to write to L would be considered its current owner.) It would maintain a directory of all caches having valid copies of that block, say C5 and C6 in our story here. As soon as P2 wrote to L, it would then send either invalidate or update packets (depending on which type was being used) to C5 and C6 (and not to other caches which didn't have valid copies).

There would also be a directory at the memory, listing the current owners of all blocks. Say for example P0 now wishes to "join the club," i.e. tries to access L, but does not have a copy of that block in its cache C0. C0 will thus not be listed in the directory for this block. So, now when it tries to access L and it will get a cache miss. P0 must now consult the **home** of L, say P14. The home might be determined by L's location in main memory according to high-order interleaving; it is the place where the main-memory version of L resides. A table at P14 will inform P0 that P2 is the current owner of that block. P0 will then send a message to P2 to add C0 to the list of caches having valid copies of that block. Similarly, a cache might "resign" from the club, due to that cache line being replaced, e.g. in a LRU setting, when some other cache miss occurs.

Example: the MESI Cache Coherency Protocol Many types of cache coherency protocols have been proposed and used, some of them quite complex. A relatively simple one for snoopy bus systems which is widely used is MESI, whose name stands for the four states a given cache line can be in for a given CPU:

- Modified
- Exclusive
- Shared
- Invalid

It is the protocol used in the Pentium I, for example.

[27] Some protocols change between the two modes dynamically.

Here is a summary of the meanings of the states:

state	meaning
M	written to more than once; no other copy valid
E	valid; no other cache copy valid; memory copy valid
S	valid; at least one other cache copy valid
I	invalid

Following is a summary of MESI state changes.[28] When reading it, keep in mind that there is a separate state for each cache/memory block combination. In other words, if we have c CPUs (and thus c caches) and b memory blocks, there are components in a state. Also, in addition to the terms **read hit**, **read miss**, **write hit**, **write miss**, which you are already familiar with, there are also **read snoop** and **write snoop**. These refer to the case in which our CPU observes a read or write action by another CPU on the bus. So, here are various events and their corresponding state changes:

If our CPU does a read:

present state	event	new state
M	read hit	M
E	read hit	E
S	read hit	S
I	read miss; no valid cache copy at any other CPU	E
I	read miss; at least one valid cache copy in some other CPU	S

If our CPU does a memory write:

present state	event	new state
M	write hit; do not put invalidate signal on bus; do not update memory	M
E	same as M above	M
S	write hit; put invalidate signal on bus; update memory	E
I	write miss; update memory but do nothing else	I

If our CPU does a read snoop (i.e. observes another CPU's cache do a read action on the bus) or **write snoop** (i.e. observes another CPU's cache to a write action on the bus):

present state	event	newstate
M	read snoop; write line back to memory, picked up by other CPU	S
M	write snoop; write line back to memory	I
E	read snoop; put shared signal on bus; no memory action	S
E	write snoop; no memory action	I
S	read snoop	S
S	write snoop	I
I	any snoop	I

[28] See *Pentium Processor System Architecture*, by D. Anderson and T. Shanley, Addison-Wesley, 1995. We have simplified the presentation here, by eliminating certain programmable options.

4.2.5 The Problem of "False Sharing"

Consider the C declaration

```
int W,Z;
```

Since **W** and **Z** are declared adjacently, most compilers will assign them contiguous memory addresses. Thus, unless one of them is at a memory block boundary, when they are cached they will be stored in the same cache line. Suppose the program writes to **Z**, and our system uses an invalidate protocol. Then **W** will be considered invalid at the other processors, even though its values at those processors' caches are correct. This is the **false sharing** problem, alluding to the fact that the two variables are sharing a cache line even though they are not related.

This can have very adverse impacts on performance. If for instance our variable **W** is now written to, then **Z** will suffer unfairly, as its copy in the cache will be considered invalid even though it is perfectly valid. This can lead to a "ping-pong" effect, in which alternate writing to two variables leads to a cyclic pattern of coherency transactions.

4.2.6 Memory-Access Consistency Policies

Though the word *consistency* in the title of this section may seem to simply be a synonym for *coherency* from the last section, and though there actually is some relation, the issues here are quite different. In this case, it is a timing issue: After one processor changes the value of a shared variable, when will that value be visible to the other processors?

There are various reasons why this is an issue. For example, many processors, especially in multiprocessor systems, have **write buffers**, which save up writes for some time before actually sending them to memory. (For the time being, let's suppose there are no caches.) The goal is to reduce memory access costs. Sending data to memory in groups is generally faster than sending one at a time, as the overhead of, for instance, acquiring the bus is amortized over many accesses. Reads following a write may proceed, without waiting for the write to get to memory, except for reads to the same address. So in a multiprocessor system in which the processors use write buffers, there will often be some delay before a write actually shows up in memory.

A related issue is that operations may occur, or appear to occur, out of order. As noted above, a read which follows a write in the program may execute before the write is sent to memory. Also, in a multiprocessor system with multiple paths between processors and memory modules, two writes might take different paths, one longer than the other, and arrive "out of order." In order to simplify the presentation here, we will focus on the case in which the problem is due to write buffers, though.

The designer of a multiprocessor system must adopt some **consistency model** regarding situations like this. The above discussion shows that the programmer must be made aware of the model, or risk getting incorrect results. Note also that different consistency models will give different levels of performance. The "weaker" consistency models make for faster machines but require the programmer to do more work.

The strongest consistency model is Sequential Consistency. It essentially requires that memory operations done by one processor are observed by the other processors to occur in the same order as executed on the

first processor. Enforcement of this requirement makes a system slow, and it has been replaced on most systems by weaker models.

One such model is **release consistency**. Here the processors' instruction sets include instructions ACQUIRE and RELEASE. Execution of an ACQUIRE instruction at one processor involves telling all other processors to flush their write buffers. However, the ACQUIRE won't execute until pending RELEASEs are done. Execution of a RELEASE basically means that you are saying, "I'm done writing for the moment, and wish to allow other processors to see what I've written." An ACQUIRE waits for all pending RELEASEs to complete before it executes.[29]

A related model is **scope consistency**. Say a variable, say **Sum**, is written to within a critical section guarded by LOCK and UNLOCK instructions.[30] Then under scope consistency any changes made by one processor to **Sum** within this critical section would then be visible to another processor when the latter next enters this critical section. The point is that memory update is postpone until it is actually needed. Also, a barrier operation (again, executed at the hardware level) forces all pending memory writes to complete.

All modern processors include instructions which implement consistency operations. For example, Sun Microsystems' SPARC has a MEMBAR instruction. If used with a STORE operand, then all pending writes at this processor will be sent to memory. If used with the LOAD operand, all writes will be made visible to this processor.

Now, how does cache coherency fit into all this? There are many different setups, but for example let's consider a design in which there is a write buffer between each processor and its cache. As the processor does more and more writes, the processor saves them up in the write buffer. Eventually, some programmer-induced event, e.g. a MEMBAR instruction,[31] will cause the buffer to be flushed. Then the writes will be sent to "memory"—actually meaning that they go to the cache, and then possibly to memory.

The point is that before that flush of the write buffer occurs, the cache coherency system is quite unaware of these writes. Thus the cache coherency operations, e.g. the various actions in the MESI protocol, won't occur until the flush happens.

To make this notion concrete, again consider the example with **Sum** above, and assume release or scope consistency. The CPU currently executing that code (say CPU 5) writes to **Sum**, which is a memory operation—it affects the cache and thus eventually the main memory—but that operation will be invisible to the cache coherency protocol for now, as it will only be reflected in this processor's write buffer. But when the unlock is finally done (or a barrier is reached), the write buffer is flushed and the writes are sent to this CPU's cache. That then triggers the cache coherency operation (depending on the state). The point is that the cache coherency operation would occur only now, not before.

What about reads? Suppose another processor, say CPU 8, does a read of **Sum**, and that page is marked invalid at that processor. A cache coherency operation will then occur. Again, it will depend on the type of coherency policy and the current state, but in typical systems this would result in **Sum**'s cache block being

[29] There are many variants of all of this, especially in the software distibuted shared memory realm, to be discussed later.

[30] Note again that these are instructions within the processors' instruction sets, not just a software operation. They would typically be made conveniently available to the programmer via library calls from C/C++, but those library functions would include LOCK and UNLOCK instructions.

[31] We call this "programmer-induced," since the programmer will include some special operation in her C/C++ code which will be translated to MEMBAR.

shipped to CPU 8 from whichever processor the cache coherency system thinks has a valid copy of the block. That processor may or may not be CPU 5, but even if it is, that block won't show the recent change made by CPU 5 to **Sum**.

The analysis above assumed that there is a write buffer between each processor and its cache. There would be a similar analysis if there were a write buffer between each cache and memory.

Note once again the performance issues. Instructions such as ACQUIRE or MEMBAR will use a substantial amount of interprocessor communication bandwidth. A consistency model must be chosen carefully by the system designer, and the programmer must keep the communication costs in mind in developing the software.

4.2.7 Fetch-and-Add and Packet-Combining Operations

Another form of interprocessor synchronization is a **fetch-and-add** (FA) instruction. The idea of FA is as follows. For the sake of simplicity, consider code like

```
LOCK(K);
Y = X++;
UNLOCK(K);
```

Suppose our architecture's instruction set included an F&A instruction. It would add 1 to the specified location in memory, and return the old value (to **Y**) that had been in that location before being incremented. And all this would be an atomic operation.

We would then replace the code above by a library call, say,

```
FETCH_AND_ADD(X,1);
```

The C code above would compile to, say,

```
F&A X,R,1
```

where **R** is the register into which the old (pre-incrementing) value of **X** would be returned.

There would be hardware adders placed at each memory module. That means that the whole operation could be done in one round trip to memory. Without F&A, we would need two round trips to memory just for the

```
X++;
```

(we would load **X** into a register in the CPU, increment the register, and then write it back to **X** in memory), and then the LOCK() and UNLOCK() would need trips to memory too. This could be a huge time savings, especially in situations like this one in which the operation is repeatedly executed in a loop.

In addition to read and write operations being specifiable in a network packet, an F&A operation could be specified as well (a 2-bit field in the packet would code which operation was desired). Again, there would be adders included at the memory modules, i.e. the addition would be done at the memory end, not at the processors. When the F&A packet arrived at a memory module, our variable **X** would have 1 added to it, while the old value would be sent back in the return packet (and put into R).

Another possibility for speedup occurs if our system uses a multistage interconnection network such as a crossbar. In that situation, we can design some intelligence into the network nodes to do **packet combining**: Say more than one CPU is executing an F&A operation at about the same time for the same variable **X**. Then more than one of the corresponding packets may arrive at the same network node at about the same time. If each one requested an incrementing of **X** by 1, the node can replace the two packets by one, with an increment of 2. Of course, this is a delicate operation, and we must make sure that different CPUs get different return values, etc.

4.2.8 Multicore Chips

A recent trend has been to put several CPUs on one chip, termed a **multicore** chip. As of March 2006, dual-core chips were beginning to become common in personal computers. The typical dual-core setup might have the two CPUs sharing a common L2 or L3 cache, and thus a common connection to the bus or interconnect network.

If our system consists of more than one of these chips, there are both challenges and opportunities. For example, if the hardware is designed well (and the software takes advantage of it), the number of systemwide cache coherency transactions may be reduced.

4.2.9 Software for Use with Shared-Memory Hardware

As mentioned earlier, programming on shared-memory systems today is typically done via threads. But to alleviate the need to do a lot of coding for low-level thread operations, a higher-level paradigm was developed cooperatively by IBM, Intel, Sun Microsystems and Hewlett-Packard, called OpenMP. It still is threads-based, but the thread creation and usage is done behind the scenes, in response to OpenMP high-level operations. For example, in OpenMP one can set up a parallel **for** loop. Each iteration of the loop will be processed by a different thread, but the programmer does not explicitly create or see the threads.

I have a longer tutorial on OpenMP at `http://heather.cs.ucdavis.edu/~matloff/openmp.html`, but OpenMP example program below is heavily-commented in order to serve as a quick tutorial. It implements the Dijkstra algorithm for finding the shortest path between two nodes in a graph.

Let G(i,j) denote the one-hop distance from vertex i to vertex j if i and j are neighbors, ∞ otherwise. Here is pseudocode describing the process on a uniprocessor system:

```
Done = {0}
NewDone = 0
NonDone = {1,2,...,N-1}
for J = 0 to N-1 Dist[J] = G(0,J)
```

```
for Step = 1 to N-1
   find J such that Dist[J] is min among all J in NonDone
   transfer J from NonDone to Done
   NewDone = J
   for K = 1 to N-1
      if K is in NonDone
         Dist[K] = min(Dist[K],Dist[NewDone]+G[NewDone,K])
```

Here is a parallel implementation in OpenMP:

```
1  // OpenMP example program:  Dijkstra shortest-path finder in a
2  // bidirectional graph
3
4  // serves as a tutorial to OpenMP; see notes in comments at the end of
5  // the file
6
7  // each thread handles one chunk of vertices
8
9  // usage:  dijkstra
10
11 #include <stdio.h>
12
13 #define LARGEINT 2<<30-1  // "infinity"
14 #define NV 6
15
16 // global variables, all shared by all threads by default
17
18 int ohd[NV][NV],  // 1-hop distances between vertices
19     mind[NV],  // min distances found so far
20     notdone[NV], // vertices not checked yet
21     nth,  // number of threads
22     chunk,  // number of vertices handled by each thread
23     md,  // current min over all threads
24     mv;  // vertex which achieves that min
25
26 void init(int ac, char **av)
27 {  int i,j;
28    for (i = 0; i < NV; i++)
29       for (j = 0; j < NV; j++)  {
30          if (j == i) ohd[i][i] = 0;
31          else ohd[i][j] = LARGEINT;
32       }
33    ohd[0][1] = ohd[1][0] = 40;
34    ohd[0][2] = ohd[2][0] = 15;
35    ohd[1][2] = ohd[2][1] = 20;
36    ohd[1][3] = ohd[3][1] = 10;
37    ohd[1][4] = ohd[4][1] = 25;
38    ohd[2][3] = ohd[3][2] = 100;
39    ohd[1][5] = ohd[5][1] = 6;
40    ohd[4][5] = ohd[5][4] = 8;
41    for (i = 1; i < NV; i++)  {
42       notdone[i] = 1;
43       mind[i] = ohd[0][i];
```

```
   }
}

// finds closest to 0 among notdone, among s through e
void findmymin(int s, int e, int *d, int *v)
{  int i;
       *d = LARGEINT;
       for (i = s; i <= e; i++)
          if (notdone[i] && mind[i] < *d)  {
             *d = ohd[0][i];
             *v = i;
          }
}

// for each i in [s,e], ask whether a shorter path to i exists, through
// mv
void updateohd(int s, int e)
{  int i;
   for (i = s; i <= e; i++)
      if (mind[mv] + ohd[mv][i] < mind[i])
         mind[i] = mind[mv] + ohd[mv][i];
}

void dowork()
{
   #pragma omp parallel  // Note 1
   {  int startv,endv,  // start, end vertices for this thread
           step,  // whole procedure goes NV steps
           mymd,  // min value found by this thread
           mymv,  // vertex which attains that value
           me = omp_get_thread_num();  // my thread number
      #pragma omp single   // Note 2
      {  nth = omp_get_num_threads();  chunk = NV/nth;
         printf("there are %d threads\n",nth);  }
      #pragma omp barrier  // Note 3
      startv = me * chunk;
      endv = startv + chunk - 1;
      for (step = 0; step < NV; step++)  {
         // find closest vertex to 0 among notdone; each thread finds
         // closest in its group, then we find overall closest
         #pragma omp single
         {  md = LARGEINT; mv = 0;  }
         findmymin(startv,endv,&mymd,&mymv);
         // update overall min if mine is smaller
         #pragma omp critical  // Note 4
         {  if (mymd < md)
              {  md = mymd; mv = mymv;  }
         }
         // mark new vertex as done
         #pragma omp barrier
         #pragma omp single
         {  notdone[mv] = 0;  }
         // now update my section of ohd
         updateohd(startv,endv);
         #pragma omp barrier
```

```
99          }
100      }
101   }
102
103   int main(int argc, char **argv)
104   {  int i;
105      init(argc,argv);
106      dowork();
107      // back to single thread now
108      printf("minimum distances:\n");
109      for (i = 1; i < NV; i++)
110         printf("%d\n",mind[i]);
111   }
112
113   // tutorial notes:
114
115   //  1.  OpenMP works via a preprocessor, which translates pragmas to
116   //      threads calls.  Note that the sharp sign ('#') must be the first
117   //      character in the line, other than blanks.
118   //
119   //      The "parallel" clause says, "Have each thread do this block"
120   //      (enclosed by braces).  Code not in a block with a "parallel"
121   //      pragma is done only by the master thread.
122
123   //  2.  The "single" clause says, "Have only one thread (whichever hits
124   //      this line first) execute the following block."
125
126   //      In this case, we are calling the OMP function
127   //      omp_get_num_threads(), which of course returns the number of
128   //      threads.  Since we assign the return value to the global variable
129   //      nth, only one thread needs to do this, so we use "single".  And
130   //      thought there would be no harm (other than a delay) if all
131   //      threads did this, in some applications we would need to limit an
132   //      action to just one thread.
133
134   //  3.  The "barrier" clause does the standard barrier operation.  Note
135   //      carefully that there are also implicit barriers following blocks
136   //      to which various OpenMP pragmas apply, such as "for" and
137   //      "single".  One can override those implicit barriers by using the
138   //      "nowait" clause.  On platforms with nonsequential memory
139   //      consistency, you can also use the "flush" directive to force a
140   //      memory update.
141
142   //  4.  The "critical" clause sets up a critical section, with invisible
143   //      lock/unlock operations.  Note carefully that the clause may be
144   //      followed by an optional name, which is crucial in some
145   //      applications.  All critical sections with the same name
146   //      are guarded by the same (invisible) locks.  Those with
147   //      no name are also guarded by the same locks, so the programmer
148   //      could really lose parallelism if he/she were not aware of this.
149
150   //      Certain very specialized one-statement critical sections can be
151   //      handled more simply and efficiently using the "atomic"
152   //      directive, e.g.
153
```

```
154 //         #pragma omp atomic
155 //         y += x;
156
157 //      Note that that statment can NOT be a block.
```

4.3 Shared-Memory through Software

4.3.1 Software Distributed Shared Memory

There are also various shared-memory software packages that run on message-passing hardware such as NOWs, called **software distributed shared memory** (SDSM) systems. Since the platforms do not have any physically shared memory, the shared-memory view which the programmer has is just an illusion. But that illusion is very useful, since the shared-memory paradigm is believed to be the easier one to program in. Thus SDSM allows us to have "the best of both worlds"—the convenience of the shared-memory world view with the inexpensive cost of some of the message-passing hardware systems, particularly networks of workstations (NOWs).

SDSM itself is divided into two main approaches, the **page-based** and **object-based** varieties. The page-based approach is generally considered clearer and easier to program in, and provides the programmer the "look and feel" of shared-memory programming better than does the object-based type.[32] We will discuss only the page-based approach here. The most popular SDSM system today is the page-based Treadmarks (Rice University). Another excellent page-based system is JIAJIA (Academy of Sciences, China).

To illustrate how page-paged SDSMs work, consider the line of JIAJIA code

```
Prime = (int *) jia_alloc(N*sizeof(int));
```

The function **jia_alloc()** is part of the JIAJIA library, **libjia.a**, which is linked to one's application program during compilation.

At first this looks a little like a call to the standard **malloc()** function, setting up an array **Prime** of size **N**. In fact, it does indeed allocate some memory. Note that each node in our JIAJIA group is executing this statement, so each node allocates some memory at that node. Behind the scenes, not visible to the programmer, each node will then have its own copy of **Prime**.

However, JIAJIA sets things up so that when one node later accesses this memory, for instance in the statement

```
Prime[I] = 1;
```

this action will eventually trigger a network transaction (not visible to the programmer) to the other JIAJIA nodes.[33] This transaction will then update the copies of **Prime** at the other nodes.[34]

[32] The term *object-based* is not related to the term *object-oriented programming*.

[33] There are a number of important issues involved with this word *eventually*, as we will see later.

[34] The update may not occur immediately. More on this later.

How is all of this accomplished? It turns out that it relies on a clever usage of the nodes' virtual memory (VM) systems. To understand this, let's review how VM systems work.

Suppose a variable **X** has the virtual address 1200, i.e. **&X = 1200**. The actual physical address may be, say, 5000. When the CPU executes a machine instruction that specifies access to 1200, the CPU will do a lookup on the **page table**, and find that the true location is 5000, and then access 5000. On the other hand, **X** may not be **resident** in memory at all, in which case the page table will say so. If the CPU finds that **X** is nonresident, it will cause an internal interrupt, which in turn will cause a jump to the operating system (OS). The OS will then read **X** in from disk,[35] place it somewhere in memory, and then update the page table to show that **X** is now someplace in memory. The OS will then execute a return from interrupt instruction,[36], and the CPU will restart the instruction which triggered the page fault.

Here is how this is exploited to develop SDSMs on Unix systems. The SDSM will call a system function such as **mprotect()**. This allows the SDSM to deliberately mark a page as nonresident (even if the page *is* resident). Basically, anytime the SDSM knows that a node's local copy of a variable is invalid, it will mark the page containing that variable as nonresident. Then, the next time the program at this node tries to access that variable, a page fault will occur.

As mentioned in the review above, normally a page fault causes a jump to the OS. However, technically any page fault in Unix is handled as a signal, specifically SIGSEGV. Recall that Unix allows the programmer to write his/her own signal handler for any signal type. In this case, that means that the programmer—meaning the people who developed JIAJIA or any other page-based SDSM—writes his/her own page fault handler, which will do the necessary network transactions to obtain the latest valid value for **X**.

Note that although SDSMs are able to create an illusion of almost all aspects of shared memory, it really is not possible to create the illusion of shared pointer variables. For example on shared memory hardware we might have a variable like **P**:

```
int Y,*P;
...
...
P = &Y;
...
```

There is no simple way to have a variable like **P** in an SDSM. This is because a pointer is an address, and each node in an SDSM has its own memory separate address space. The problem is that even though the underlying SDSM system will keep the various copies of **Y** at the different nodes consistent with each other, **Y** will be at a potentially different address on each node.

All SDSM systems must deal with a software analog of the cache coherency problem. Whenever one node modifies the value of a shared variable, that node must notify the other nodes that a change has been made. The designer of the system must choose between update or invalidate protocols, just as in the hardware case.[37] Recall that in non-bus-based shared-memory multiprocessors, one needs to maintain a directory

[35] Actually, it will read the entire page containing **X** from disk, but to simplify language we will just say **X** here.

[36] E.g. **iret** on Pentium chips.

[37] Note, though, that we are not actually dealing with a cache here. Each node in the SDSM system will have a cache, of course, but a node's cache simply stores parts of that node's set of pages. The coherency across nodes is across pages, not caches. We must insure that a change made to a given page is eventually propropagated to pages on other nodes which correspond to this one.

which indicates at which processor a valid copy of a shared variable exists. Again, SDSMs must take an approach similar to this.

Similarly, each SDSM system must decide between sequential consistency, release consistency etc. More on this later.

Note that in the NOW context the internode communication at the SDSM level is typically done by TCP/IP network actions. Treadmarks uses UDP, which is faster than TCP. but still part of the slow TCP/IP protocol suite. TCP/IP was simply not designed for this kind of work. Accordingly, there have been many efforts to use more efficient network hardware and software. The most popular of these is the Virtual Interface Architecture (VIA).

Not only are coherency actions more expensive in the NOW SDSM case than in the shared-memory hardware case due to network slowness, there is also expense due to granularity. In the hardware case we are dealing with cache blocks, with a typical size being 512 bytes. In the SDSM case, we are dealing with pages, with a typical size being 4096 bytes. The overhead for a cache coherency transaction can thus be large.

4.3.2 Case Study: JIAJIA

Programmer Interface

We will not go into detail on JIAJIA programming here. There is a short tutorial on JIAJIA at `http://heather.cs.ucdavis.edu/~matloff/jiajia.html`, but here is an overview:

- one writes in C/C++[38], making call to the JIAJIA library, which is linked in upon compilation
- the library calls include standard shared-memory operations for lock, unlock, barrier, processor number, etc., plus some calls aimed at improving performance

Following is a JIAJIA example program, performing Odd/Even Transposition Sort. This is a variant on Bubble Sort, sometimes useful in parallel processing contexts.[39] The algorithm consists of n phases, in which each processor alternates between trading with its left and right neighbors.

```
1  // JIAJIA example program:  Odd-Even Tranposition Sort
2
3  // array is of size n, and we use n processors; this would be more
4  // efficient in a "chunked" versions, of course (and more suited for a
5  // message-passing context anyway)
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <jia.h>  // required include; also link via -ljia
10
11 // pointer to shared variable
```

[38] Or FORTRAN.

[39] Though, as mentioned in the comments, it is aimed more at message-passing contexts.

```
12 int *x;  // array to be sorted
13
14 int n,  // range to check for primeness
15     debug;  // 1 for debugging, 0 else
16
17 // if first arg is bigger, then replace it by the second
18 void cpsmaller(int *p1,int *p2)
19 {  int tmp;
20         if (*p1 > *p2)  *p1 = *p2;
21 }
22
23 // if first arg is smaller, then replace it by the second
24 void cpbigger(int *p1,int *p2)
25 {  int tmp;
26         if (*p1 < *p2)  *p1 = *p2;
27 }
28
29 // does sort of m-element array y
30 void oddeven(int *y, int m)
31 {  int i,left=jiapid-1,right=jiapid+1,newval;
32    for (i=0; i < m; i++)  {
33       if ((i+jiapid)%2 == 0)  {
34          if (right < m)
35             if (y[jiapid] > y[right]) newval = y[right];
36       }
37       else  {
38          if (left >= 0)
39             if (y[jiapid] < y[left]) newval = y[left];
40       }
41       jia_barrier();
42       if ((i+jiapid)%2 == 0 && right < m || (i+jiapid)%2 == 1 && left >= 0)
43             y[jiapid] = newval;
44       jia_barrier();
45    }
46 }
47
48 main(int argc, char **argv)
49 {  int i,mywait=0;
50    jia_init(argc,argv);  // required init call
51    // get command-line arguments (shifted for nodes > 0)
52    if (jiapid == 0)  {
53       n = atoi(argv[1]);
54       debug = atoi(argv[2]);
55    }
56    else  {
57       n = atoi(argv[2]);
58       debug = atoi(argv[3]);
59    }
60    jia_barrier();
61    // create a shared array x of length n
62    x = (int *) jia_alloc(n*sizeof(int));
63    // barrier recommended after allocation
64    jia_barrier();
65    // node 0 gets simple test array from command-line
66    if (jiapid == 0)  {
```

```
67          for (i = 0; i < n; i++)
68             x[i] = atoi(argv[i+3]);
69       }
70       jia_barrier();
71       if (debug && jiapid == 0)
72          while (mywait == 0)  { ; }
73       jia_barrier();
74       oddeven(x,n);
75       if (jiapid == 0)  {
76          printf("\nfinal array\n");
77          for (i = 0; i < n; i++)
78             printf("%d\n",x[i]);
79       }
80       jia_exit();
81    }
```

System Workings

JIAJIA's main characteristics as an SDSM are:

- page-based
- scope consistency
- home-based
- multiple writers

Let's take a look at these.

As mentioned earlier, one first calls **jia_alloc()** to set up one's shared variables. Note that this will occur at each node, so there are multiple copies of each variable; the JIAJIA system ensures that these copies are consistent with each other, though of course subject to the laxity afforded by scope consistency.

Recall that under scope consistency, a change made to a shared variable at one processor is guaranteed to be made visible to another processor if the first processor made the change between lock/unlock operations and the second processor accesses that variable between lock/unlock operations on that same lock.[40]

Each page—and thus each shared variable—has a **home** processor. If another processor writes to a page, then later when it reaches the unlock operation it must send all changes it made to the page back to the home node. In other words, the second processor calls **jia_unlock()**, which sends the changes to its sister invocation of **jia_unlock()** at the home processor.[41] Say later a third processor calls **jia_lock()** on that same lock, and then attempts to read a variable in that page. A page fault will occur at that processor, resulting in the JIAJIA system running, which will then obtain that page from the first processor.

[40] Writes will also be propagated at barrier operations, but two successive arrivals by a processor to a barrier can be considered to be a lock/unlock pair, by considering a departure from a barrier to be a "lock," and considering reaching a barrier to be an "unlock." So, we'll usually not mention barriers separately from locks in the remainder of this subsection.

[41] The set of changes is called a **diff**, remiscent of the Unix file-compare command. A copy, called a **twin**, had been made of the original page, which now will be used to produce the diff. This has substantial overhead. The Treadmarks people found that it took 167 microseconds to make a twin, and as much as 686 microseconds to make a diff.

Note that all this means the JIAJIA system at each processor must maintain a page table, listing where each home page resides.[42] At each processor, each page has one of three states: Invalid, Read-Only, Read-Write. State changes, though, are reported when lock/unlock operations occur. For example, if CPU 5 writes to a given page which had been in Read-Write state at CPU 8, the latter will not hear about CPU 5's action until some CPU does a lock. This CPU need not be CPI 8. When one CPU does a lock, it must coordinate with all other nodes, at which time state-change messages will be piggybacked onto lock-coordination messages.

Note also that JIAJIA allows the programmer to specify which node should serve as the home of a variable, via one of several forms of the **jia_alloc()** call. The programmer can then tailor his/her code accordingly. For example, in a matrix problem, the programmer may arrange for certain rows to be stored at a given node, and then write the code so that most writes to those rows are done by that processor.

The general principle here is that writes performed at one node can be made visible at other nodes on a "need to know" basis. If for instance in the above example with CPUs 5 and 8, CPU 2 does not access this page, it would be wasteful to send the writes to CPU 2, or for that matter to even inform CPU 2 that the page had been written to. This is basically the idea of all non-Sequential consistency protocols, even though they differ in approach and in performance for a given application.

JIAJIA allows multiple writers of a page. Suppose CPU 4 and CPU 15 are simultaneously writing to a particular page, and the programmer has relied on a subsequent barrier to make those writes visible to other processors.[43] When the barrier is reached, each will be informed of the writes of the other.[44] Allowing multiple writers helps to reduce the performance penalty due to false sharing.

5 Program Performance Issues

5.1 The Problem

In one's first experience with parallel processing, say with 16 processors, one might naively expect to see one's program run about 16 times faster. The reality, though, is the following:

- there are many significant sources of overhead to slow things down
- because of that overhead, one may find that parallel processing yields a substantial speed advantage over sequential processing only on very large problems; in smaller problems, a parallel program may actually be considerably slower than a sequential one
- the programmer must devote considerable effort to write the code in such a way as to mitigate the overhead
- the type of interconnect between nodes has a huge impact on the potential speedup from parallel processing

[42] In JIAJIA, that location is normally fixed, but JIAJIA does include advanced programmer options which allow the location to migrate.

[43] The only other option would be to use lock/unlock, but then their writing would not be simultaneous.

[44] If they are writing to the same variable, not just the same page, the programmer would use locks instead of a barrier, and the situation would not arise.

5.2 Some Timing Comparisons

David J. Pursley of the Dept. of Computer Science at Bucknell University did some timing experiments on parallel Mergesort, summarized below.[45]

The two types of sort were compared to a sequential (i.e. one-node) quicksort. The version used for the latter was optimized, with no recursion and with the sorting done "in place," i.e. without other arrays being used for intermediate steps.

First, the experiment was run on what Pursley refers to as a "network," presumably a LAN. Here are the results, measured in seconds:

	Two-Way Merge		Three-Way Merge		Quicksort
NUMPROC->	3	7	3	7	1
NUM2SORT					
1008	5.50	6.07	3.05	4.82	0.04
2016	9.37	12.06	6.47	9.56	0.10
4032	22.54	28.50	14.03	19.57	0.24

As you can see, parallel operations really slowed us down! The communication overhead is simply too high.

He then modified the algorithm so that each node sends an entire array at once, instead of one data item at a time. He used PVM as his programming language, and ran this one on a hypercube. Here are the results:

	Two-Way Merge		Three-Way Merge		Quicksort
NUMPROC->	3	7	3	7	1
NUM2SORT					
1008	0.06	0.18	0.06	0.09	0.04
2016	0.12	0.23	0.14	0.15	0.10
4032	0.25	0.35	0.18	0.28	0.24
8064	0.51	0.55	0.36	0.47	0.44
16128	1.01	1.11	0.71	0.95	0.98
32256	2.05	2.14	1.44	1.96	2.13
64512	4.26	4.23	3.00	3.16	4.84
129024	9.08	9.05	5.99	6.21	9.72
258048	18.91	17.88	12.41	12.41	21.95
516096	37.56	38.40	25.24	25.00	42.87

So, for small arrays quicksort is either better or similar to the parallel sorts, but for the much larger arrays we do see a definite improvement. On the other hand, we are not seeing very good scalability—e.g. a speedup of less than 2 for 7 processors.

Note that by sending an entire array at once we are actually reducing parallelism, so the speedup we see in the latter table may seem paradoxical. But the reason for it is that the software overhead to send and receive a message is very high; by sending whole arrays at once, we reduce that overhead per data item.

[45] The results are rather old by now, but they still illustrate the issues well.

5.3 Solutions

One must always keep in mind the communication costs. For a NOW, for instance, Myrinet has lower communication costs than Ethernet, but the latency on Myrinet is on the order of microseconds, far slower than local memory access. For a shared-memory machine, a crossbar provides the most paths but also has a slow latency, as there is a delay at each switch in a path.

Much of this overhead can be mitigated in various ways, such as:

- in message-passing settings, keep as many of the operations on the local node as possible
- in shared-memory NUMA settings, keep as many of the operations on the local memory as possible
- in shared-memory settings, both hardware and SDSM, try to minimize cache coherency transactions
- in message-passing settings, try to make most messages long ones, saving up data if necessary, in order to amortize the latency costs
- in shared-memory settings, both hardware and SDSM, note the memory consistency model, and take advantage of it (again in an amortization sense)

5.4 Time Measurement

Since speed is of the essence in most parallel processing applications, an issue arises as to how to measure run time.

In Unix, a first-level measure is the **time** command. If the program is named, say, **a.out**, we type

```
$ time a.out
```

This causes **a.out** to run, and a report as to time spent follows. The output includes the time the program spend in user mode, system mode (for OS calls) and **wall clock time**, i.e. the actual time elapsed. Note carefully that the user time is the sum of such times for all threads of the program.

A better approach, though, is to use timing functions available in your parallel processing tools package. In MPI, one can call **MPI_Wtime()**. The corresonding function in OpenMP is **omp_get_wtime()**.

6 Debugging Multicomputer Programs

Even in nonparallel contexts, you should never debug by using **printf()** or **cout**. It is just very distracting and wasteful of time to keep inserting **printf()/cout** lines, recompiling, etc.[46] Instead, use a debugging tool. I will use GDB as an example here.[47]

[46] And in the parallel case, it is even worse, because the output to **stdout** from the various nodes is typically all mixed together.

[47] Note that DDD provides a GUI interface to GDB and various other debuggers. You may find it more useful than direct use of GDB.

Use the debugging tool to follow my Principle of Confirmation: Keep checking that key variables have the values you think they should, and that the program's flow of control follows the path you think it should. (Use the debugging tool to do the checking.) Eventually you will find a situation where your expectation fails; at that point you will have discovered the rough location of the bug, and can reason out what is wrong.

In the parallel case, use of a debugging tool becomes more complex, because you need to have a separate invocation of the tool for each node. For concreteness, let's use as our example the MPI program in Section 2.3. (The same principles hold for shared-memory programming.) It runs on three nodes, say **pc10**, **pc11** and **pc12**. Here is what you need to do:

- Start the program running at **pc10** from one window.
- In another window, log in to **pc10** and run the Unix command **ps ax**,[48] to determine the process number of your MPI program on this node. Then type

  ```
  gdb program_name process_number
  ```

  GDB will then join the program in progress.[49]
- In two other windows, do the same at **pc11** and **pc12**.[50]

A problem that arises here is that there will of course be some delay while you are determining process numbers and invoking GDB. By that time, the program may be finished, or at least may be past the place at which you had intended to set a breakpoint. The standard way of dealing with this is to set a variable, which we named **DebugWait** in the example in Section 2.3, which forces the program to stop and give us humans a chance to "catch up":

```
while (DebugWait) ;
```

If we wish to debug, we specify a value of 1 for **DebugWait** on the command line.[51] The program will stop there, and then when we get there with GDB, we can change the value of the variable to 0, thus causing the program to resume execution when we issue GDB's **continue** command.

We need not manually intervene in all nodes. In MPI, for instance, we may have node 0 wait for manual intevention, and then broadcast a go-ahead message to the other nodes.[52] In a shared-memory context, say JIAJIA, one could accomplish the same effect using a barrier; CPU 0 would wait for manual intervention, and the other nodes would wait for CPU 0 at a barrier.

Note that any output may be delayed. Use **flush()** or **fflush()** to ensure that everything gets output right away.

[48] It may be a slightly different command on some flavors of Unix.

[49] You may prefer to already have GDB running and then use GDB's **attach** command to attach the process. This way you do not have to exit GDB between runs of your program, and you retain breakpoints and other GDB state.

[50] You may find that using three separate windows requires too much space on your monitor. An alternative would be to use the **screen** program, available on most Unix systems. Within a single window, you can have multiple shells, toggling among them by hitting a specified key sequence. This way you use monitor space for only one window. Thanks to Bryan Richter for this suggestion.

[51] See the line **DebugWait = atoi(Argv[2])** in the program's source code.

[52] Thanks for M. Wiley for this suggestion.

7 Barrier Implementation

Recall that a **barrier** is program code[53] which has a processor do a wait-loop action until all processors have reached that point in the program.

A function **Barrier()** is often supplied as a library function; here we will see how to implement such a function.[54]

In our implementation of **Barrier()**, suppose we will have available some library functions like SYS_SIZE and CPU_NUM, which give us the number of CPUs and the CPU number, respectively.

7.1 A Use-Once Version

```
1  struct BarrStruct {
2     int Count,Lock;
3  } ;
4
5  Barrier(struct BarrStruct *PB)
6  {  LOCK(PB->Lock);
7     PB->Count++;
8     UNLOCK(PB->Lock);
9     while (PB->Count < SYS_SIZE) ;
10 }
```

This is very simple, actually overly so. This implementation will work once, so if the example above didn't have two calls to **Barrier()** it would be fine, but not otherwise.

What is the problem? Clearly, something must be done to reset **count** to 0 at the end of the call, but doing this safely is not so easy, as seen in the next subsection.

7.2 An Attempt to Write a Reusable Version

Consider the following attempt at fixing the code for **barrier()**:

```
1  struct BarrStruct {
2     int Count;
3     int Lock;
4  } ;
5
6  Barrier(struct BarrStruct *PB;
7  {  int OldCount;
8
9     LOCK(PB->Lock);
10    OldCount = PB->Count++;
```

[53] Some hardware barriers have been proposed.

[54] Again, keep in mind that this is a library function, not a system call. We are not relying on the OS here.

```
11     UNLOCK(PB->Lock);
12     if (OldCount == SYS_SIZE-1)  PB->Count = 0;
13     else while (PB->Count < SYS_SIZE) ;
14  }
```

Unfortunately, this doesn't work either. To see why, consider a loop with a barrier call at the end:

```
1   struct BarrStruct B;  /* global variable! */
2   ........
3   while(.......)  {
4      .........
5      Barrier(&B);
6      .........
7   }
```

At the end of the first iteration of the loop, all the processors will wait at the barrier until everyone catches up. After this happens, one processor, say 12, will reset **B.Count** to 0, as desired. But if we are unlucky, some other processor, say processor 3, will perform the second iteration of the loop in an extremely short period of time, and will reach the barrier and increment the Count variable before processor 12 resets it. This would result in disaster, since processor 3's increment would be canceled, leaving us one short when we try to finish the barrier the second time.[55]

One way to avoid this would be to have *two* Count variables, and have the processors alternate using one then the other. In the scenario described above, processor 3 would increment the *other* Count variable, and thus would not conflict with processor 12's resetting. Here is a safe barrier function based on this idea:

```
1   struct BarrStruct {
2      int Count[2];
3      int Lock;
4      int EvenOdd;
5   } ;
6
7   Barrier(struct BarrStruct *PB)
8   {  int Par,OldCount;
9      Par = PB->EvenOdd;
10     LOCK(PB->Lock);
11     OldCount = PB->Count[Par]++;
12     UNLOCK(PB->Lock);
13     if (OldCount == SYS_SIZE-1) PB->Count[Par] = 0;
14     else while (PB->Count[Par] > 0) ;
15     PB->EvenOdd = 1 - Par;
16  }
```

[55] Another disaster scenario which might occur, of course, is that one processor might reset **B.Count** to 0 before another processor had a chance to notice that **B.Count** had reached SYS_SIZE.