Programming on Parallel Machines

Norman Matloff
University of California, Davis [1]

[1] **Licensing:** This work is licensed under a Creative Commons Attribution-No Derivative Works 3.0 United States License. Copyright is retained by N. Matloff in all non-U.S. jurisdictions, but permission to use these materials in teaching is still granted, provided the authorship and licensing information here is displayed in each unit. I would appreciate being notified if you use this book for teaching, just so that I know the materials are being put to use, but this is not required.

Author's Biographical Sketch

Dr. Norm Matloff is a professor of computer science at the University of California at Davis, and was formerly a professor of statistics at that university. He is a former database software developer in Silicon Valley, and has been a statistical consultant for firms such as the Kaiser Permanente Health Plan.

Dr. Matloff was born in Los Angeles, and grew up in East Los Angeles and the San Gabriel Valley. He has a PhD in mathematics from UCLA, specializing in probability and statistics. His current research interests are parallel processing, statistical analysis of social networks, and statistical regression methodology.

Prof. Matloff is a former appointed member of IFIP Working Group 11.3, an international committee concerned with statistical database security, established under UNESCO. He was a founding member of the UC Davis Department of Statistics, and participated in the formation of the UCD Computer Science Department as well. He is a recipient of the Distinguished Teaching Award at UC Davis.

Dr. Matloff is the author of two published textbooks, and of a number of widely-used Web tutorials on computer topics, such as the Linux operating system and the Python programming language. He and Dr. Peter Salzman are authors of *The Art of Debugging with GDB, DDD, and Eclipse*. Prof. Matloff's book on the R programming language, *The Art of R Programming*, is due to be published in 2010. He is also the author of several open-source textbooks, including *From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science* (`http://heather.cs.ucdavis.edu/probstatbook`), and *Programming on Parallel Machines* (`http://heather.cs.ucdavis.edu/~matloff/ParProcBook.pdf`).

Contents

Chapter 1

Introduction to Parallel Processing

Parallel machines provide a wonderful opportunity for applications with large computational requirements. Effective use of these machines, though, requires a keen understanding of how they work. This chapter provides an overview.

1.1 Overview: Why Use Parallel Systems?

1.1.1 Execution Speed

There is an ever-increasing appetite among some types of 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. 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. The database field has contributed greatly to the commercial success of large shared-memory machines.

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.

Note, though, that it may be much more challenging than this implies. First of all, the computation will need some communication between the processors, which hinders performance if it is not done carefully. Second, if one really wants good speedup, one may need to take into account the fact that some squares require more computation work than others. More on this below.

In this setting you need the program to run as fast as possible. Thus, in order to write good parallel processing software, you must have a good knowledge of the underlying hardware. You must find 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.

1.1.2 Memory

Yes, execution speed is the reason that comes to most people's minds when the subject of parallel processing comes up. But in many applications, an equally important consideration is memory capacity. Parallel processing application often tend to use huge amounts of memory, and in many cases the amount of memory needed is more than can fit on one machine. If we have many machines working together, especially in the message-passing settings described below, we can accommodate the large memory needs.

1.2 Parallel Processing Hardware

This is not a hardware course, but since the goal of using parallel hardware is speed, the efficiency of our code is a major issue. That in turn means that we need a good understanding of the underlying hardware that we are programming. In this section, we give an overview of parallel hardware.

1.2.1 Shared-Memory Systems

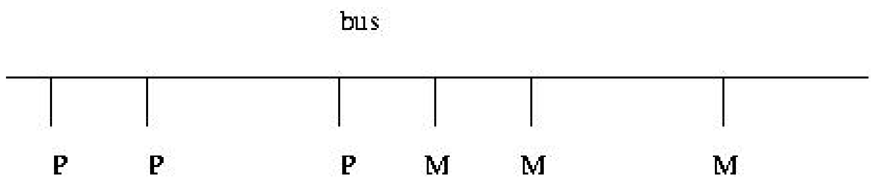
1.2.1.1 Basic Architecture

Here many CPUs share the same physical memory. This kind of architecture is sometimes called MIMD, standing for Multiple Instruction (different CPUs are working independently, and thus typically are executing different instructions at any given instant), Multiple Data (different CPUs are generally accessing different memory locations at any given time).

Until recently, shared-memory systems cost hundreds of thousands of dollars and were affordable only by large companies, such as in the insurance and banking industries. The high-end machines are indeed still quite expensive, but now **dual-core** machines, in which two CPUs share a common memory, are commonplace in the home.

1.2.1.2 Example: SMP Systems

A Symmetric Multiprocessor (SMP) system has the following structure:



Here and below:

- 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 memory modules as processors. In the shared-memory case, the memory modules collectively form the entire shared address space, but with the addresses being assigned to the memory modules 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.

    We need 10 bits for addresses (since $1024 = 2^{10}$). The two most-significant bits would be used to select the module number (since $4 = 2^2$); hence the term *high-order* in the name of this design. The remaining eight bits are used to select the word within a module.

  - (b)

    Low-order interleaving. Here consecutive addresses are in consecutive memory modules (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.

    Here the two least-significant bits are used to determine the module number.

- To make sure only one processor 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.

1.2.2 Message-Passing Systems

1.2.2.1 Basic Architecture

Here we have a number of independent CPUs, each with its own independent memory. The various processors communicate with each other via networks of some kind.

1.2.2.2 Example: Networks of Workstations (NOWs)

Large shared-memory multiprocessor systems are still very expensive. A major alternative today is networks of workstations (NOWs). Here one purchases a set of commodity PCs and networks them for use as parallel processing systems. The PCs are of course individual machines, capable of the usual uniprocessor (or now multiprocessor) 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 in Chapter 6. But even without these techniques, the price/performance ratio in NOW is much superior in many applications to that of shared-memory hardware.

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

NOWs have become so popular that there are now “recipes” on how to build them for the specific purpose of parallel processing. The term **Beowulf** come to mean a cluster of PCs, usually with a fast network connecting them, used for parallel processing. Software packages such as ROCKS (`http://www.rocksclusters.org/wordpress/`) have been developed to make it easy to set up and administer such systems.

1.2.3 SIMD

In contrast to MIMD systems, processors in SIMD—Single Instruction, Multiple Data—systems execute in lockstep. At any given time, all processors are executing the same machine instruction on different data.

Some famous SIMD systems in computer history include the ILLIAC and Thinking Machines Corporation's CM-1 and CM-2. Also, DSP ("digital signal processing") chips tend to have an SIMD architecture.

But today the most prominent example of SIMD is that of GPUs—graphics processing units. In addition to powering your PC's video cards, GPUs can now be used for general-purpose computation. The architecture is fundamentally shared-memory, but the individual processors do execute in lockstep, SIMD-fashion.

1.3 Programmer 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.

1.3.1 Shared-Memory

1.3.1.1 Programmer View

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.

1.3.1.2 Example

Today, programming on shared-memory multiprocessors is typically done via **threading**. (Or, as we will see in other chapters, by higher-level code that runs threads underneath.) 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,[1] Windows, Python, Java and Perl all support threaded programming.

In the typical implementation, a thread is a special case of an OS process. One important difference is that the various threads of a program share memory. (One can arrange for processes to share memory too in some OSs, but they don't do so by default.)

On a uniprocessor system, the threads of a program take turns executing, so that there is only an illusion of parallelism. But on a multiprocessor system, one can genuinely have threads running in parallel.

One of the most popular threads systems is Pthreads, whose name is short for POSIX threads. POSIX is a Unix standard, and the Pthreads system was designed to standardize threads programming on Unix. It has since been ported to other platforms.

Following is an example of Pthreads programming, in which we determine the number of prime numbers in a certain range. Read the comments at the top of the file for details; the threads operations will be explained presently.

```
// PrimesThreads.c

// threads-based program to find the number of primes between 2 and n;
// uses the Sieve of Eratosthenes, deleting all multiples of 2, all
// multiples of 3, all multiples of 5, etc.

// for illustration purposes only; NOT claimed to be efficient

// Unix compilation:  gcc -g -o primesthreads PrimesThreads.c -lpthread -lm

// usage:  primesthreads n num_threads

#include <stdio.h>
#include <math.h>
#include <pthread.h>  // required for threads usage

#define MAX_N 100000000
#define MAX_THREADS 25

// shared 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
// lock for the shared variable nextbase
pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
// ID structs for the threads
pthread_t id[MAX_THREADS];

// "crosses out" all odd multiples of k
void crossout(int k)
{  int i;
   for (i = 3; i*k <= n; i += 2)  {
```

[1]Here and below, the term *Unix* includes Linux.

```
      prime[i*k] = 0;
   }
}

// each thread runs this routine
void *worker(int tn)  // tn is the thread number (0,1,...)
{  int lim,base,
       work = 0;  // amount of work done by this thread
   // no need to check multipliers bigger than sqrt(n)
   lim = sqrt(n);
   do  {
      // get next sieve multiplier, avoiding duplication across threads
      // lock the lock
      pthread_mutex_lock(&nextbaselock);
      base = nextbase;
      nextbase += 2;
      // unlock
      pthread_mutex_unlock(&nextbaselock);
      if (base <= lim)  {
         // don't bother crossing out if base known composite
         if (prime[base])  {
            crossout(base);
            work++;  // log work done by this thread
         }
      }
      else return work;
   } while (1);
}

main(int argc, char **argv)
{  int nprimes,  // number of primes found
       i,work;
   n = atoi(argv[1]);
   nthreads = atoi(argv[2]);
   // mark all even numbers nonprime, and the rest "prime until
   // shown otherwise"
   for (i = 3; i <= n; i++)  {
      if (i%2 == 0) prime[i] = 0;
      else prime[i] = 1;
   }
   nextbase = 3;
   // get threads started
   for (i = 0; i < nthreads; i++)  {
      // this call says to create a thread, record its ID in the array
      // id, and get the thread started executing the function worker(),
      // passing the argument i to that function
      pthread_create(&id[i],NULL,worker,i);
   }

   // wait for all done
   for (i = 0; i < nthreads; i++)  {
      // this call said to wait until thread number id[i] finishes
      // execution, and to assign the return value of that thread to our
      // local variable work here
      pthread_join(id[i],&work);
      printf("%d values of base done\n",work);
   }

   // report results
   nprimes = 1;
   for (i = 3; i <= n; i++)
      if (prime[i])  {
         nprimes++;
      }
   printf("the number of primes found was %d\n",nprimes);

}
```

To make our discussion concrete, suppose we are running this program with two threads. Suppose also the both threads are running simultaneously most of the time. This will occur if they aren't competing for turns with other big threads, say 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 threads is via global variables, so if they are evil, they are a necessary evil. Personally I think the stern admonitions against global variables is overblown anyway. See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.

As mentioned earlier, the globals are shared by all processors.[2] 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.

Note that in the statement

```
pthread_mutex_t nextbaselock = PTHREAD_MUTEX_INITIALIZER;
```

the right-hand side is not a constant. It is a macro call, and is thus something which is executed.

In the code

```
pthread_mutex_lock(&nextbaselock);
base = nextbase
```

[2]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.

```
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 "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
- 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 11, duplicating work and thus 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 that these problems could occur either on a uniprocessor or multiprocessor system. In the uniprocessor case, thread A's turn might end right after it reads **nextbase**, followed by a turn by B which executes that same instruction. In the multiprocessor case, A and B could literally be running simultaneously, but still with the action by B coming an instant after A.

This problem frequently arises in parallel database systems. 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.

A critical section is always a potential bottlement in a parallel program, because its code is serial instead of parallel. In our program here, we may get better performance by having each thread work on, say, five values of **nextbase** at a time. Our line

```
nextbase += 2;
```

would become

```
nextbase += 10;
```

That would mean that any given thread would need to go through the critical section only one-fifth as often, thus greatly reducing overhead. On the other hand, near the end of the run, this may result in some threads being idle while other threads still have a lot of work to do.

Note this code.

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

This is a special case of of **barrier**.

A barrier is a point in the code that all threads must reach before continuing. In this case, a barrier is needed in order to prevent premature execution of the later code

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

which would result in possibly wrong output if we start counting primes before some threads are done.

The **pthread_join()** function actually causes the given thread to exit, so that we then “join” the thread that created it, i.e. **main()**. Thus some may argue that this is not really a true barrier.

Barriers are very common in shared-memory programming, and will be discussed in more detail in Chapter 2.

1.3.2 Message Passing

1.3.2.1 Programmer View

By contrast, in the message-passing paradigm, all nodes would have separate copies of A, X and Y. 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.

1.3.3 Example

Here we use the MPI system, with our hardware being a NOW.

MPI is 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.

```

/* 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 */

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

#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]);

   /* this loop is here to synchronize all nodes for debugging;
      if DebugWait is specified as 1 on the command line, all nodes
      wait here until the debugging programmer starts GDB at all
      nodes and within GDB sets DebugWait to 0 to then proceed */
   while (DebugWait) ;

   /* mandatory to begin any MPI program */
   MPI_Init(&Argc,&Argv);

   /* puts the number of nodes in NNodes */
   MPI_Comm_size(MPI_COMM_WORLD,&NNodes);
   /* puts the node number of this node in Me */
   MPI_Comm_rank(MPI_COMM_WORLD,&Me);

   /* OK, get started; first record current time in T1 */
   if (Me == 2) T1 = MPI_Wtime();
}

Node0()

{  int I,Dummy,
       Error;  /* not checked in this example */
   for (I = 1; I <= N/2; I++)  {
      ToCheck = 2 * I + 1;
      if (ToCheck > N) break;
      /* MPI_Send  --  send a message
               parameters:
                pointer to place where message is to be drawn from
                number of items in message
                item type
                destination node
                message type ("tag") programmer-defined
                node group number (in this case all nodes) */
      if (ToCheck % 3 > 0)
         Error = MPI_Send(&ToCheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
   }
   Error = MPI_Send(&Dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
}

Node1()

{  int Error,  /* not checked in this example */
       Dummy;
   MPI_Status Status;  /* see below */

   while (1)  {
      /* MPI_Recv  --  receive a message
                parameters:
                pointer to place to store message
                number of items in message (see notes on
                   this at the end of this file)
                item type
                accept message from which node(s)
                message type ("tag"), programmer-defined (in this
          case any type)
                node group number (in this case all nodes)
                status (see notes on this at the end of this file) */
      Error = MPI_Recv(&ToCheck,1,MPI_INT,0,MPI_ANY_TAG,
                  MPI_COMM_WORLD,&Status);
      if (Status.MPI_TAG == END_MSG) break;
      if (ToCheck % 5 > 0)
         Error = MPI_Send(&ToCheck,1,MPI_INT,2,PIPE_MSG,MPI_COMM_WORLD);
   }
   /* now send our end-of-data signal, which is conveyed in the
      message type, not the message (we have a dummy message just
      as a placeholder */
   Error = MPI_Send(&Dummy,1,MPI_INT,2,END_MSG,MPI_COMM_WORLD);
}

Node2()

{  int ToCheck,  /* current number to check from Node 0 */
       Error,  /* not checked in this example */
       PrimeCount,I,IsComposite;
   MPI_Status Status;  /* see below */

   PrimeCount = 3;  /* must account for the primes 2, 3 and 5, which
                       won't be detected below */
   while (1)  {
      Error = MPI_Recv(&ToCheck,1,MPI_INT,1,MPI_ANY_TAG,
                  MPI_COMM_WORLD,&Status);
      if (Status.MPI_TAG == END_MSG) break;
      IsComposite = 0;
      for (I = 7; I*I <= ToCheck; I += 2)
         if (ToCheck % I == 0)  {
```

```
      IsComposite = 1;
      break;
   }
      if (!IsComposite) PrimeCount++;
   }
   /* check the time again, and subtract to find run time */
   T2 = MPI_Wtime();
   printf("elapsed time = %f\n",(float)(T2-T1));
   /* print results */
   printf("number of primes = %d\n",PrimeCount);
}

main(argc,argv)
   int argc; char **argv;

{  Init(argc,argv);
   /* note:  instead of having a switch statement, we could write
      three different programs, each running on a different node */
   switch (Me)  {
      case 0:  Node0();
               break;
      case 1:  Node1();
               break;
      case 2:  Node2();
   };
   /* mandatory for all MPI programs */
   MPI_Finalize();
}

/* explanation of "number of items" and "status" arguments at the end
   of MPI_Recv():

   when receiving a message you must anticipate the longest possible
   message, but the actual received message may be much shorter than
   this; you can call the MPI_Get_count() function on the status
   argument to find out how many items were actually received

   the status argument will be a pointer to a struct, containing the
   node number, message type and error status of the received
   message

   say our last parameter is Status; then Status.MPI_SOURCE
   will contain the number of the sending node, and
   Status.MPI_TAG will contain the message type; these are
   important if used MPI_ANY_SOURCE or MPI_ANY_TAG in our
   node or tag fields but still have to know who sent the
   message or what kind it is */
```

The set of machines can be heterogeneous, but MPI “translates” for you automatically. If say one node has a big-endian CPU and another has a little-endian CPU, MPI will do the proper conversion.

1.4 Relative Merits: Shared-Memory Vs. Message-Passing

It is generally believed in the parallel processing community that the shared-memory paradigm produces code that is easier to write, debug and maintain than message-passing.

On the other hand, in some cases message-passing can produce faster code. Consider the Odd/Even Transposition Sort algorithm, for instance. Here pairs of processes repeatedly swap sorted arrays with each other. In a shared-memory setting, this might produce a bottleneck at the shared memory, slowing down the code. Of course, the obvious solution is that if you are using a shared-memory machine, you should just choose some other sorting algorithm, one tailored to the shared-memory setting.

There used to be a belief that message-passing was more **scalable**, i.e. amenable to very large systems. However, GPU has demonstrated that one can achieve extremely good scalability with shared-memory.

My own preference, obviously, is shared-memory.

Chapter 2

Shared Memory Parallelism

Shared-memory programming is considered by many in the parallel processing community as being the clearest of the various parallel paradigms available.

2.1 What Is Shared?

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. All of the stacks are in shared memory, but they will be accessed separately, since each CPU has a different value in its SP (Stack Pointer) register. 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 that writes to **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 a local variable **Y** to something like ESP+8, the third item on the stack (on a 32-bit machine), 8 bytes past the word pointed to by the stack pointer, ESP. The OS will assign a different ESP value to each thread, so the stacks of the various threads will be separate. Each CPU has its own ESP register, containing the location of the stack for whatever thread that CPU is currently running. So, the value of **Y** will be different for each thread.

2.2 Structures for Sharing

2.2.1 Memory Modules

Parallel execution of a program requires, to a large extent, parallel accessing of memory. To some degree this is handled by having a cache at each CPU, but it is also facilitated by dividing the memory into separate modules. This way several memory accesses can be done simultaneously.

This raises the question of how to divide up the memory into modules. There are two main ways to do this:

(a) High-order interleaving. Here consecutive addresses are in the <u>same</u> 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.

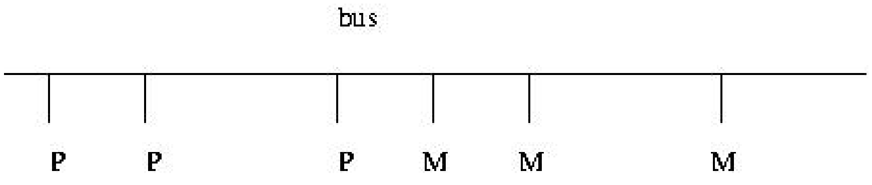
(b) Low-order interleaving. Here consecutive addresses are in consecutive M's (except when we get to the right end). In the example above, if we used low-order interleaving, then address 0 would be in M0, 1 would be in M1, 2 would be in M2, 3 would be in M3, 4 would be back in M0, 5 in M1, and so on.

Say we will have eight modules. Then under high-order interleaving, the first two bits of a word's address would be taken to be the module number, with the remaining bits being address within module. Under low-order interleaving, the two least significant bits would be used.

Low-order interleaving is often used for **vector processors**. On such a machine, we might have both a regular add instruction, ADD, and a vector version, VADD. The latter would add two vectors together, so it would need to read two vectors from memory. If low-order interleaving is used, the elments of these vectors are spread across the various modules, so fast access is possible.

2.2.2 SMP Systems

A Symmetric Multiprocessor (SMP) system has the following structure:



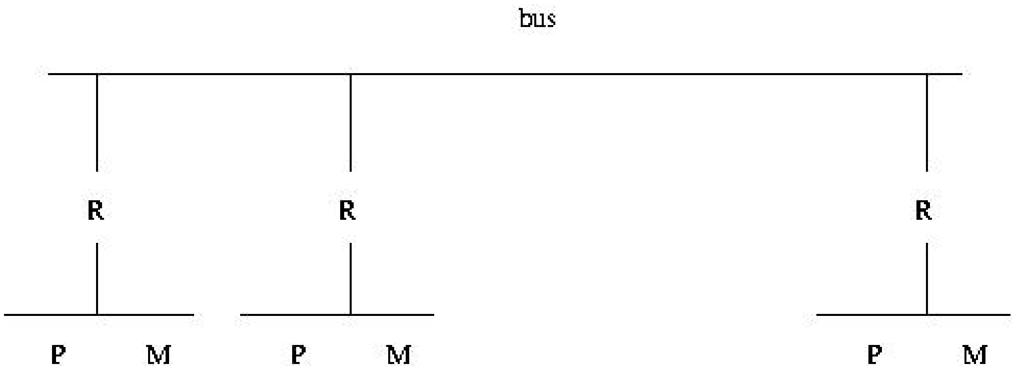
Here and below:

- 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.
- 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.

2.2.3 NUMA Systems

In a **Nonuniform Memory Access** (NUMA) architecture, each CPU has a memory module physically next to it, and these processor/memory (P/M) pairs are connected by some kind of network.

Here is a simple 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.

Suppose for example that P3 needs to access location 200, and suppose that high-order interleaving is used. If location 200 is in M3, then P3's request is satisfied by the local bus.[1] 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.

Today almost all high-end MIMD systems are NUMAs. 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.

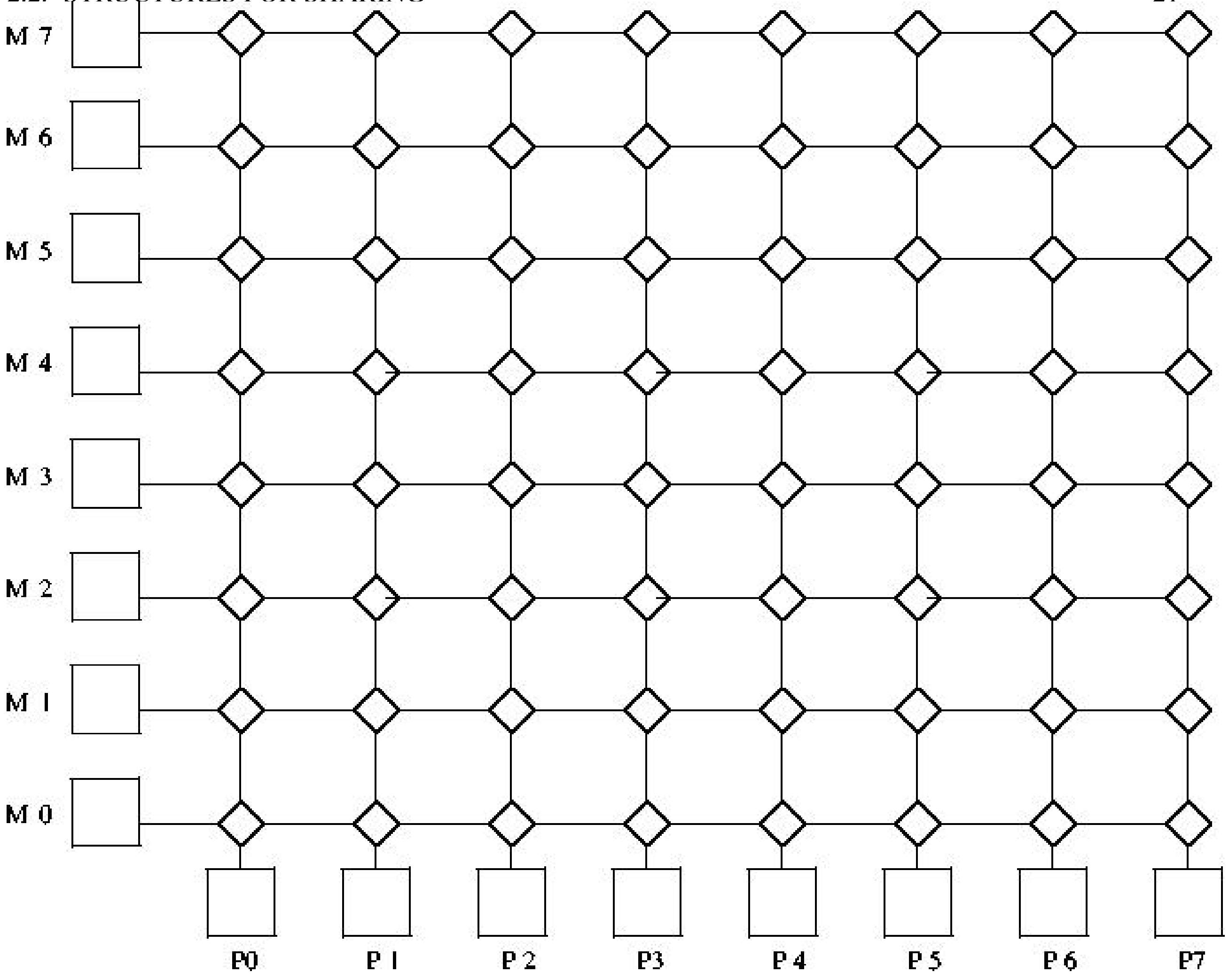
2.2.4 NUMA Interconnect Topologies

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.

2.2.4.1 Crossbar Interconnects

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

[1]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.

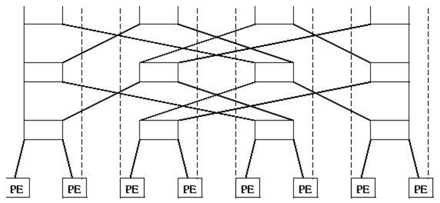
There could also be a return network of the same type, with this one being memory → processor, to return the result of the read requests.[2]

Another 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, similar to the Omega network shown below.

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.

2.2.4.2 Omega (or Delta) 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.[3]

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

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

[3]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.

from the left input).

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

$$m = (2k + \lfloor (2k)/N \rfloor)\ mod\ N \tag{2.1}$$

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

2.2.5 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)	$\mathrm{O}(\log_2 \mathrm{n})$	O(n)
bandwidth	O(1)	O(n)	O(n)
cost	O(1)	$\mathrm{O}(\mathrm{n} \log_2 \mathrm{n})$	$\mathrm{O}(\mathrm{n}^2)$

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).[4]

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.

[4] 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.

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

That is not 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.

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.

2.2.6 Why Have Memory in Modules?

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.

- (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.

The idea is to have several modules busy at once, say in conjunction with a **split-transaction bus**. Here, after a processor makes a memory request, it relinquishes the bus, allowing others to use it while the memory does the requested work. Without splitting the memory into modules, this wouldn't achieve parallelism. The bus does need extra lines to identify which processor made the request.

2.3 Test-and-Set Type Instructions

Consider a bus-based system. In addition to whatever memory read and memory write instructions the processor included, there would also be a TAS instruction.[5] 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.

On Pentium machines, the LOCK prefix can be used to get atomicity for certain instructions.[6] For example,

```
lock add $2, x
```

would add the constant 2 to the memory location labeled **x** in an atomic manner.

The LOCK prefix locks the bus for the entire duration of the instruction. Note that the ADD instruction here involves two memory transactions—one to read the old value of **x**, and the second the write the new, incremented value back to **x**. So, we are locking for a rather long time, but the benefits can be huge.

A good example of this kind of thing would be our program **PrimesThreads.c** in Chapter 1, where our critical section consists of adding 2 to **nextbase**. There we surrounded the add-2 code by Pthreads lock

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

[6]The instructions ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD. Also, XCHG asserts the LOCK# bus signal even if the LOCK prefix is specified. Locking only applies to these instructions in forms in which there is an operand in memory.

and unlock operations. These involve system calls, which are very time consuming, involving hundreds of machine instructions. Compare that to the one-instruction solution above! The very heavy overhead of pthreads would be thus avoided.

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).

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 (e.g. the bus in a bus-based 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.

2.4 Cache Issues

2.4.1 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.[7] 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

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

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 actually doing the write) tells each other cache to mark as invalid its cache line (if any) which contains a copy of the variable.[8] 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, along with P5. Then P5 and P6 may have to contend with each other. Say P6 manages to grab possession of the bus first.[9] P6 then executes the TAS again, which finds L = 0 and changes L back to 1. P6 then relinquishes the bus, and enters the critical section. Note that in changing L to 1, P6 also sends an invalidate signal to all the

[8]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.

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

other caches. So, when P5 tries its execution of the TAS again, it will have to ask P6 to send a valid copy of the block. P6 does so, but L will be 1, 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[10] 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 bus/network 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.)

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.[11]

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 <u>positive</u> feature in terms of cache coherency, because it makes <u>broadcast</u> 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 <u>not</u> to other caches which didn't have valid copies).

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

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

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.

2.4.2 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, which for example is the protocol used in the Pentium series.

MESI is an invalidate protocol for bus-based systems. Its name stands for the four states a given cache line can be in for a given CPU:

- Modified
- Exclusive
- Shared
- Invalid

Note that *each memory block* has such a state at *each cache*. For instance, block 88 may be in state S at P5's and P12's caches but in state I at P1's cache.

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 (block either not in the cache or present but incorrect)

Following is a summary of MESI state changes.[12] When reading it, keep in mind again that there is a separate state for each cache/memory block combination.

[12]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.

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 on the bus a block request by another CPU that has attempted a read or write action but encountered a miss in its own cache; if our cache has a valid copy of that block, we must provide it to the requesting CPU (and in some cases to memory).

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 or write snoop:

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, signal other CPU now OK to do its write	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

Note that a write miss does NOT result in the associated block being brought in from memory.

Example: Suppose a given memory block has state M at processor A but has state I at processor B, and B attempts to write to the block. B will see that its copy of the block is invalid, so it notifies the other CPUs via the bus that it intends to do this write. CPU A sees this announcement, tells B to wait, writes its own copy of the block back to memory, and then tells B to go ahead with its write. The latter action means that A's copy of the block is not correct anymore, so the block now has state I at A. B's action does not cause loading of that block from memory to its cache, so the block still has state I at B.

2.4.3 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.

2.5 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.[13]

A related model is **scope consistency**. Say a variable, say **Sum**, is written to within a critical section guarded by LOCK and UNLOCK instructions. 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 postponed 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,[14] 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 (in this type of setup) 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.

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

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

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 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.

The recent Pentium models use Sequential Consistency, with any write done by a processor being immediately sent to its cache as well.

2.6 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 for long-latency interconnects.

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.

2.7 Multicore Chips

A recent trend has been to put several CPUs on one chip, termed a **multicore** chip. As of March 2008, dual-core chips are common in personal computers, and quad-core machines are within reach of the budgets of many people. Just as the invention of the integrated circuit revolutionized the computer industry by making computers affordable for the average person, multicore chips will undoubtedly revolutionize the world of parallel programming.

A typical dual-core setup might have the two CPUs sharing a common L2 cache, with each CPU having its own L3 cache. The chip may interface to the bus or interconnect network of via an L1 cache.

Multicore is extremely important these days. However, they are just SMPs, for the most part, and thus should not be treated differently.

2.8 Illusion of Shared-Memory through Software

2.8.0.1 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.[15] 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.[16] This transaction will then update the copies of **Prime** at the other nodes.[17]

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

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

[17]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,[18] 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,[19], 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

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

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

case.[20] Recall that in non-bus-based shared-memory multiprocessors, one needs to maintain a directory 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.

2.8.0.2 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++ (or FORTRAN), making calls 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.[21] The algorithm consists of n phases, in which each processor alternates between trading with its left and right neighbors.

```
// JIAJIA example program:  Odd-Even Tranposition Sort

// array is of size n, and we use n processors; this would be more
```

[20]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.

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

```
// efficient in a "chunked" versions, of course (and more suited for a
// message-passing context anyway)

#include <stdio.h>
#include <stdlib.h>
#include <jia.h>  // required include; also must link via -ljia

// pointer to shared variable
int *x;  // array to be sorted

int n,  // range to check for primeness
    debug;  // 1 for debugging, 0 else

// if first arg is bigger, then replace it by the second
void cpsmaller(int *p1,int *p2)
{  int tmp;
        if (*p1 > *p2)  *p1 = *p2;
}

// if first arg is smaller, then replace it by the second
void cpbigger(int *p1,int *p2)
{  int tmp;
        if (*p1 < *p2)  *p1 = *p2;
}

// does sort of m-element array y
void oddeven(int *y, int m)
{  int i,left=jiapid-1,right=jiapid+1,newval;
   for (i=0; i < m; i++)  {
      if ((i+jiapid)%2 == 0)  {
         if (right < m)
            if (y[jiapid] > y[right]) newval = y[right];
      }
      else  {
         if (left >= 0)
            if (y[jiapid] < y[left]) newval = y[left];
      }
      jia_barrier();
      if ((i+jiapid)%2 == 0 && right < m || (i+jiapid)%2 == 1 && left >= 0)
            y[jiapid] = newval;
      jia_barrier();
   }
}

main(int argc, char **argv)
{  int i,mywait=0;
   jia_init(argc,argv);  // required init call
   // get command-line arguments (shifted for nodes > 0)
   if (jiapid == 0)  {
      n = atoi(argv[1]);
      debug = atoi(argv[2]);
   }
   else  {
      n = atoi(argv[2]);
      debug = atoi(argv[3]);
   }
   jia_barrier();
   // create a shared array x of length n
   x = (int *) jia_alloc(n*sizeof(int));
   // barrier recommended after allocation
   jia_barrier();
   // node 0 gets simple test array from command-line
   if (jiapid == 0)  {
      for (i = 0; i < n; i++)
         x[i] = atoi(argv[i+3]);
   }
   jia_barrier();
   if (debug && jiapid == 0)
      while (mywait == 0)  { ; }
   jia_barrier();
   oddeven(x,n);
   if (jiapid == 0)  {
      printf("\nfinal array\n");
      for (i = 0; i < n; i++)
         printf("%d\n",x[i]);
   }
   jia_exit();
}
```

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.[22]

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

[22]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.

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.[23] 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.

Note that all this means the JIAJIA system at each processor must maintain a page table, listing where each home page resides.[24] 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.[25] When the barrier is reached, each will be informed of the writes of the other.[26] Allowing multiple writers helps to reduce the performance penalty due to false sharing.

[23]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.

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

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

[26]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.

2.9 Barrier Implementation

Recall that a **barrier** is program code[27] which has a processor do a wait-loop action until all processors have reached that point in the program.[28]

A function **Barrier()** is often supplied as a library function; here we will see how to implement such a library function in a correct and efficient manner. Note that since a barrier is a serialization point for the program, efficiency is crucial to performance.

Implementing a barrier in a fully correct manner is actually a bit tricky. We'll see here what can go wrong, and how to make sure it doesn't.

In this section, we will approach things from a shared-memory point of view. But the methods apply in the obvious way to message-passing systems as well, as will be discused later.

2.9.1 A Use-Once Version

```
struct BarrStruct {
   int NNodes, // number of threads participating in the barrier
       Count, // number of threads that have hit the barrier so far
       pthread_mutex_t Lock = PTHREAD_MUTEX_INITIALIZER;
} ;

Barrier(struct BarrStruct *PB)
{  pthread_mutex_lock(&PB->Lock);
   PB->Count++;
   pthread_mutex_unlock(&PB->Lock);
   while (PB->Count < PB->NNodes) ;
}
```

This is very simple, actually overly so. This implementation will work once, so if a program using it doesn't make two calls to **Barrier()** it would be fine. But not otherwise. If, say, there is a call to **Barrier()** in a loop, we'd be in trouble.

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 section.

[27]Some hardware barriers have been proposed.

[28]I use the word *processor* here, but it could be just a thread on the one hand, or on the other hand a processing element in a message-passing context.

2.9.2 An Attempt to Write a Reusable Version

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

```
Barrier(struct BarrStruct *PB)
{  int OldCount;
   pthread_mutex_lock(&PB->Lock);
   OldCount = PB->Count++;
   pthread_mutex_unlock(&PB->Lock);
   if (OldCount == PB->NNodes-1)  PB->Count = 0;
   while (PB->Count < PB->NNodes) ;
}
```

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

```
struct BarrStruct B;  // global variable
........
while (.......)  {
   .........
   Barrier(&B);
   .........
}
```

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 then race ahead, perform the second iteration of the loop in an extremely short period of time, and then reach the barrier and increment the **Count** variable before processor 12 resets it to 0. 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.

Another disaster scenario which might occur is that one processor might reset **B.Count** to 0 before another processor had a chance to notice that **B.Count** had reached **B.NNodes**.

2.9.3 A Correct Version

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:

```
struct BarrStruct {
   int NNodes, // number of threads participating in the barrier
       Count[2], // number of threads that have hit the barrier so far
       pthread_mutex_t Lock = PTHREAD_MUTEX_INITIALIZER;
} ;

Barrier(struct BarrStruct *PB)
{  int Par,OldCount;
   Par = PB->EvenOdd;
   pthread_mutex_lock(&PB->Lock);
   OldCount = PB->Count[Par]++;
   pthread_mutex_unlock(&PB->Lock);
   if (OldCount == PB->NNodes-1)  {
      PB->Count[Par] = 0;
      PB->EvenOdd = 1 - Par;
   }
   else while (PB->Count[Par] > 0) ;
}
```

2.9.4 Refinements

2.9.4.1 Use of Wait Operations

The code

```
else while (PB->Count[Par] > 0) ;
```

is harming performance, since it has the processor spining around doing no useful work. In the Pthreads context, we can use a condition variable:

```
struct BarrStruct {
   int NNodes, // number of threads participating in the barrier
       Count[2], // number of threads that have hit the barrier so far
       pthread_mutex_t Lock = PTHREAD_MUTEX_INITIALIZER;
       pthread_cond_t CV = PTHREAD_COND_INITIALIZER;
} ;

Barrier(struct BarrStruct *PB)
{  int Par,I;
   Par = PB->EvenOdd;
   pthread_mutex_lock(&PB->Lock);
   PB->Count[Par]++;
   if (PB->Count < PB->NNodes)
      pthread_cond_wait(&PB->CV,&PB->Lock);
   else  {
      PB->Count[Par] = 0;
      PB->EvenOdd = 1 - Par;
      for (I = 0; I < PB->NNodes-1; I++)
         pthread_cond_signal(&PB->CV);
   }
   pthread_mutex_unlock(&PB->Lock);
}
```

Here, if a thread finds that not everyone has reached the barrier yet, it still waits for the rest, but does so passively, via the wait for the condition variable **CV**. This way the thread is not wasting valuable time on that processor, which can run other useful work.

Note that the call to **pthread_cond_wait()** requires use of the lock. Your code must lock the lock before making the call. The call itself immediately unlocks that lock after it registers the wait with the threads manager. But the call blocks until awakened when another thread calls **pthread_cond_signal()** or **pthread_cond_broadcast()**.

It is required that your code lock the lock before calling **pthread_cond_signal()**, and that it unlock the lock after the call.

By using **pthread_cond_wait()** and placing the unlock operation later in the code, as seen above, we actually could get by with just a single **Count** variable, as before.

Even better, the **for** loop could be replaced by a single call

```
pthread_cond_broadcast(&PB->PB->CV);
```

This still wakes up the waiting threads one by one, but in a much more efficient way, and it makes for clearer code.

2.9.4.2 Parallelizing the Barrier Operation

2.9.4.2.1 Tree Barriers It is clear from the code above that barriers can be costly to performance, since they rely so heavily on critical sections, i.e. serial parts of a program. Thus in many settings it is worthwhile to parallelize not only the general computation, but also the barrier operations themselves.

Consider for instance a barrier in which 16 threads are participating. We could speed things up by breaking this barrier down into two sub-barriers, with eight threads each. We would then set up three barrier operations: one of the first group of eight threads, another for the other group of eight threads, and a third consisting of a "competition" between the two groups. The variable **NNodes** above would have the value 8 for the first two barriers, and would be equal to 2 for the third barrier.

Here thread 0 could be the representative for the first group, with thread 4 representing the second group. After both groups's barriers were hit by all of their members, threads 0 and 4 would participated in the third barrier.

Note that then the notification phase would the be done in reverse: When the third barrier was complete, threads 0 and 4 would notify the members of their groups.

This would parallelize things somewhat, as critical-section operations could be executing simultaneously for the first two barriers. There would still be quite a bit of serial action, though, so we may wish to do further splitting, by partitioning each group of four threads into two subroups of two threads each.

In general, for n threads (with n, say, equal to a power of 2) we would have a tree structure, with $log_2 n$ levels in the tree. The i^{th} level (starting with the root as level 0) with consist of 2^i parallel barriers, each one representing $n/2^i$ threads.

2.9.4.2.2 Butterfly Barriers Another method basically consists of each node "shaking hands" with every other node. In the shared-memory case, handshaking could be done by having a global array **ReachedBarrier**. When thread 3 and thread 7 shake hands, for instance, would reach the barrier, thread 3 would set **ReachedBarrier[3]** to 1, and would then wait for **ReachedBarrier[7]** to become 1. The wait, as before, could either be a **while** loop or a call to **pthread_cond_wait()**. Thread 7 would do the opposite.

If we have n nodes, again with n being a power of 2, then the barrier process would consist of $log_2 n$ phases, which we'll call phase 0, phase 1, etc. Then the process works as follows.

For any node i, let i(k) be the number obtained by inverting bit k in the binary representation of i, with bit 0 being the least significant bit. Then in the k^{th} phase, node i would shake hands with node i(k).

For example, say n = 8. In phase 0, node $5 = 101_2$, say, would shake hands with node $4 = 100_2$.

Actually, a butterfly exchange amounts to a number of simultaneously tree operations.

Chapter 3

The Python Threads and Multiprocessing Modules

Python's thread system builds on the underlying OS threads. Thus are thus pre-emptible. Note, though, that Python adds its own threads manager on top of the OS thread system; see Section 3.3.

3.1 Python Threads Modules

Python threads are accessible via two modules, **thread.py** and **threading.py**. The former is more primitive, thus easier to learn from, and we will start with it.

3.1.1 The `thread` Module

The example here involves a client/server pair.[1] As you'll see from reading the comments at the start of the files, the program does nothing useful, but is a simple illustration of the principles. We set up two invocations of the client; they keep sending letters to the server; the server concatenates all the letters it receives.

Only the server needs to be threaded. It will have one thread for each client.

Here is the client code, **clnt.py**:

```
# simple illustration of thread module

```

[1]It is preferable here that the reader be familiar with basic network programming. See my tutorial at `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`. However, the comments preceding the various network calls would probably be enough for a reader without background in networks to follow what is going on.

```
# two clients connect to server; each client repeatedly sends a letter,
# stored in the variable k, which the server appends to a global string
# v, and reports v to the client; k = '' means the client is dropping
# out; when all clients are gone, server prints the final string v

# this is the client; usage is

#    python clnt.py server_address port_number

import socket  # networking module
import sys

# create Internet TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = sys.argv[1]  # server address
port = int(sys.argv[2])  # server port

# connect to server
s.connect((host, port))

while(1):
   # get letter
   k = raw_input('enter a letter:')
   s.send(k)  # send k to server
   # if stop signal, then leave loop
   if k == '': break
   v = s.recv(1024)  # receive v from server (up to 1024 bytes)
   print v

s.close() # close socket
```

And here is the server, **srvr.py**:

```
# simple illustration of thread module

# multiple clients connect to server; each client repeatedly sends a
# letter k, which the server adds to a global string v and echos back
# to the client; k = '' means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server

import socket  # networking module
import sys

import thread

# note the globals v and nclnt, and their supporting locks, which are
#    also global; the standard method of communication between threads is
#    via globals

# function for thread to serve a particular client, c
def serveclient(c):
   global v,nclnt,vlock,nclntlock
   while 1:
      # receive letter from c, if it is still connected
      k = c.recv(1)
      if k == '': break
      # concatenate v with k in an atomic manner, i.e. with protection
      #    by locks
      vlock.acquire()
      v += k
      vlock.release()
      # send new v back to client
      c.send(v)
   c.close()
   nclntlock.acquire()
   nclnt -= 1
   nclntlock.release()

# set up Internet TCP socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

port = int(sys.argv[1])  # server port number
# bind lstn socket to this port
lstn.bind(('', port))
# start listening for contacts from clients (at most 2 at a time)
lstn.listen(5)

# initialize concatenated string, v
v = ''
# set up a lock to guard v
vlock = thread.allocate_lock()

# nclnt will be the number of clients still connected
nclnt = 2
# set up a lock to guard nclnt
nclntlock = thread.allocate_lock()

# accept calls from the clients
for i in range(nclnt):
   # wait for call, then get a new socket to use for this client,
   #    and get the client's address/port tuple (though not used)
   (clnt,ap) = lstn.accept()
   # start thread for this client, with serveclient() as the thread's
   #    function, with parameter clnt; note that parameter set must be
   #    a tuple; in this case, the tuple is of length 1, so a comma is
   #    needed
   thread.start_new_thread(serveclient,(clnt,))

# shut down the server socket, since it's not needed anymore
lstn.close()

# wait for both threads to finish
while nclnt > 0: pass

print 'the final value of v is', v
```

Make absolutely sure to run the programs before proceeding further.[2] Here is how to do this:

[2]You can get them from the **.tex** source file for this tutorial, located wherever your picked up the **.pdf** version.

I'll refer to the machine on which you run the server as **a.b.c**, and the two client machines as **u.v.w** and **x.y.z**.[3] First, on the server machine, type

```
python srvr.py 2000
```

and then on each of the client machines type

```
python clnt.py a.b.c 2000
```

(You may need to try another port than 2000, anything above 1023.)

Input letters into both clients, in a rather random pattern, typing some on one client, then on the other, then on the first, etc. Then finally hit Enter without typing a letter to one of the clients to end the session for that client, type a few more characters in the other client, and then end that session too.

The reason for threading the server is that the inputs from the clients will come in at unpredictable times. At any given time, the server doesn't know which client will sent input next, and thus doesn't know on which client to call **recv()**. One way to solve this problem is by having threads, which run "simultaneously" and thus give the server the ability to read from whichever client has sent data.[4].

So, let's see the technical details. We start with the "main" program.[5]

```
vlock = thread.allocate_lock()
```

Here we set up a **lock variable** which guards **v**. We will explain later why this is needed. Note that in order to use this function and others we needed to import the **thread** module.

```
nclnt = 2
nclntlock = thread.allocate_lock()
```

We will need a mechanism to insure that the "main" program, which also counts as a thread, will be passive until both application threads have finished. The variable **nclnt** will serve this purpose. It will be a count of how many clients are still connected. The "main" program will monitor this, and wrap things up later when the count reaches 0.

```
thread.start_new_thread(serveclient,(clnt,))
```

[3]You could in fact run all of them on the same machine, with address name **localhost** or something like that, but it would be better on separate machines.

[4]Another solution is to use nonblocking I/O. See this example in that context in `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`

[5]Just as you should write the main program first, you should read it first too, for the same reasons.

Having accepted a a client connection, the server sets up a thread for serving it, via **thread.start_new_thread()**. The first argument is the name of the application function which the thread will run, in this case **serveclient()**. The second argument is a tuple consisting of the set of arguments for that application function. As noted in the comment, this set is expressed as a tuple, and since in this case our tuple has only one component, we use a comma to signal the Python interpreter that this is a tuple.

So, here we are telling Python's threads system to call our function **serveclient()**, supplying that function with the argument **clnt**. The thread becomes "active" immediately, but this does not mean that it starts executing right away. All that happens is that the threads manager adds this new thread to its list of threads, and marks its current state as Run, as opposed to being in a Sleep state, waiting for some event.

By the way, this gives us a chance to show how clean and elegant Python's threads interface is compared to what one would need in C/C++. For example, in **pthreads**, the function analogous to **thread.start_new_thread()** has the signature

```
pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
   void *(*thread_function)(void *), void *arguments);
```

What a mess! For instance, look at the types in that third argument: A pointer to a function whose argument is pointer to `void` and whose value is a pointer to `void` (all of which would have to be `cast` when called). It's such a pleasure to work in Python, where we don't have to be bothered by low-level things like that.

Now consider our statement

```
while nclnt > 0: pass
```

The statement says that as long as at least one client is still active, do nothing. Sounds simple, and it is, but you should consider what is really happening here.

Remember, the three threads—the two client threads, and the "main" one—will take turns executing, with each turn lasting a brief period of time. Each time "main" gets a turn, it will loop repeatedly on this line. But all that empty looping in "main" is wasted time. What we would really like is a way to prevent the "main" function from getting a turn at all until the two clients are gone. There are ways to do this which you will see later, but we have chosen to remain simple for now.

Now consider the function **serveclient()**. Any thread executing this function will deal with only one particular client, the one corresponding to the connection **c** (an argument to the function). So this **while** loop does nothing but read from that particular client. If the client has not sent anything, the thread will block on the line

```
k = c.recv(1)
```

This thread will then be marked as being in Sleep state by the thread manager, thus allowing the other client thread a chance to run. If neither client thread can run, then the "main" thread keeps getting turns. When a user at one of the clients finally types a letter, the corresponding thread unblocks, i.e. the threads manager changes its state to Run, so that it will soon resume execution.

Next comes the most important code for the purpose of this tutorial:

```
vlock.acquire()
v += k
vlock.release()
```

Here we are worried about a **race condition**. Suppose for example **v** is currently 'abx', and Client 0 sends **k** equal to 'g'. The concern is that this thread's turn might end in the middle of that addition to **v**, say right after the Python interpreter had formed 'abxg' but before that value was written back to **v**. This could be a big problem. The next thread might get to the same statement, take **v**, still equal to 'abx', and append, say, 'w', making **v** equal to 'abxw'. Then when the first thread gets its next turn, it would finish its interrupted action, and set **v** to 'abxg'—which would mean that the 'w' from the other thread would be lost.

All of this hinges on whether the operation

```
v += k
```

is interruptible. Could a thread's turn end somewhere in the midst of the execution of this statement? If not, we say that the operation is **atomic**. If the operation were atomic, we would not need the lock/unlock operations surrounding the above statement. I did this, using the methods described in Section 3.3.4.1, and it appears to me that the above statement is *not* atomic.

Moreover, it's safer not to take a chance, especially since Python compilers could vary or the virtual machine could change; after all, we would like our Python source code to work even if the machine changes.

So, we need the lock/unlock operations:

```
vlock.acquire()
v += k
vlock.release()
```

The lock, **vlock** here, can only be held by one thread at a time. When a thread executes this statement, the Python interpreter will check to see whether the lock is locked or unlocked right now. In the latter case, the interpreter will lock the lock and the thread will continue, and will execute the statement which updates **v**. It will then release the lock, i.e. the lock will go back to unlocked state.

If on the other hand, when a thread executes **acquire()** on this lock when it is locked, i.e. held by some other thread, its turn will end and the interpreter will mark this thread as being in Sleep state, waiting for the lock to be unlocked. When whichever thread currently holds the lock unlocks it, the interpreter will change the blocked thread from Sleep state to Run state.

Note that if our threads were non-preemptive, we would not need these locks.

Note also the crucial role being played by the global nature of **v**. Global variables are used to communicate between threads. In fact, recall that this is one of the reasons that threads are so popular—easy access to global variables. Thus the dogma so often taught in beginning programming courses that global variables must be avoided is wrong; on the contrary, there are many situations in which globals are necessary and natural.[6]

The same race-condition issues apply to the code

```
nclntlock.acquire()
nclnt -= 1
nclntlock.release()
```

Following is a Python program that finds prime numbers using threads. Note carefully that it is not claimed to be efficient at all (it may well run more slowly than a serial version); it is merely an illustration of the concepts. Note too that we are using the simple **thread** module, rather than **threading**.

```
#!/usr/bin/env python

import sys
import math
import thread

def dowork(tn):  # thread number tn
   global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
   donelock[tn].acquire()
   nstartedlock.acquire()
   nstarted += 1
   nstartedlock.release()
   lim = math.sqrt(n)
   nk = 0
   while 1:
      nextilock.acquire()
      k = nexti
      nexti += 1
      nextilock.release()
      if k > lim: break
      nk += 1
      if prime[k]:
         r = n / k
         for i in range(2,r+1):
            prime[i*k] = 0
   print 'thread', tn, 'exiting; processed', nk, 'values of k'
   donelock[tn].release()
```

[6] I think that dogma is presented in a far too extreme manner anyway. See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.

```
28
29 def main():
30    global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
31    n = int(sys.argv[1])
32    prime = (n+1) * [1]
33    nthreads = int(sys.argv[2])
34    nstarted = 0
35    nexti = 2
36    nextilock = thread.allocate_lock()
37    nstartedlock = thread.allocate_lock()
38    donelock = []
39    for i in range(nthreads):
40       d = thread.allocate_lock()
41       donelock.append(d)
42       thread.start_new_thread(dowork,(i,))
43    while nstarted < nthreads: pass
44    for i in range(nthreads):
45       donelock[i].acquire()
46    print 'there are', reduce(lambda x,y: x+y, prime) - 2, 'primes'
47
48 if __name__ == '__main__':
49     main()
```

So, let's see how the code works.

The algorithm is the famous Sieve of Erathosthenes: We list all the numbers from 2 to **n**, then cross out all multiples of 2 (except 2), then cross out all multiples of 3 (except 3), and so on. The numbers which get crossed out are composite, so the ones which remain at the end are prime.

Line 32: We set up an array **prime**, which is what we will be "crossing out." The value 1 means "not crossed out," so we start everything at 1. (Note how Python makes this easy to do, using list "multiplication.")

Line 33: Here we get the number of desired threads from the command line.

Line 34: The variable **nstarted** will show how many threads have already started. This will be used later, in Lines 43-45, in determining when the **main()** thread exits. Since the various threads will be writing this variable, we need to protect it with a lock, on Line 37.

Lines 35-36: The variable **nexti** will say which value we should do "crossing out" by next. If this is, say, 17, then it means our next task is to cross out all multiples of 17 (except 17). Again we need to protect it with a lock.

Lines 39-42: We create the threads here. The function executed by the threads is named **dowork()**. We also create locks in an array **donelock**, which again will be used later on as a mechanism for determining when **main()** exits (Line 44-45).

Lines 43-45: There is a lot to discuss here.

To start, recall that in **srvr.py**, our example in Section 3.1.1, we didn't want the main thread to exit until the child threads were done.[7] So, Line 50 was a **busy wait**, repeatedly doing nothing (**pass**). That's a waste of time—each time the main thread gets a turn to run, it repeatedly executes **pass** until its turn is over.

Here in our primes program, a premature exit by **main()** result in printing out wrong answers. On the other hand, we don't want **main()** to engage in a wasteful busy wait. We could use **join()** from **threading.Thread** for this purpose, as we have before, but here we take a different tack: We set up a list of locks, one for each thread, in a list **donelock**. Each thread initially acquires its lock (Line 9), and releases it when the thread finishes its work (Lin 27). Meanwhile, **main()** has been waiting to acquire those locks (Line 45). So, when the threads finish, **main()** will move on to Line 46 and print out the program's results.

But there is a subtle problem (threaded programming is notorious for subtle problems), in that there is no guarantee that a thread will execute Line 9 before **main()** executes Line 45. That's why we have a busy wait in Line 43, to make sure all the threads acquire their locks before **main()** does. Of course, we're trying to avoid busy waits, but this one is quick.

Line 13: We need not check any "crosser-outers" that are larger than $\sqrt{n}$.

Lines 15-25: We keep trying "crosser-outers" until we reach that limit (Line 20). Note the need to use the lock in Lines 16-19. In Line 22, we check the potential "crosser-outer" for primeness; if we have previously crossed it out, we would just be doing duplicate work if we used this **k** as a "crosser-outer."

Here's one more example, a type of Web crawler. This one continually monitors the access time of the Web, by repeatedly accessing a list of "representative" Web sites, say the top 100. What's really different about this program, though, is that we've reserved one thread for human interaction. The person can, whenever he/she desires, find for instance the mean of recent access times.

```
import sys
import time
import os
import thread

class glbls:
   acctimes = []  # access times
   acclock = thread.allocate_lock()  # lock to guard access time data
   nextprobe = 0  # index of next site to probe
   nextprobelock = thread.allocate_lock()  # lock to guard access time data
   sites = open(sys.argv[1]).readlines()  # the sites to monitor
   ww = int(sys.argv[2])  # window width

def probe(me):
   if me > 0:
      while 1:
         # determine what site to probe next
         glbls.nextprobelock.acquire()
         i = glbls.nextprobe
         i1 = i + 1
         if i1 >= len(glbls.sites): i1 = 0
```

[7]The effect of the main thread ending earlier would depend on the underlying OS. On some platforms, exit of the parent may terminate the child threads, but on other platforms the children continue on their own.

```
         glbls.nextprobe = i1
         glbls.nextprobelock.release()
         # do probe
         t1 = time.time()
         os.system('wget --spider -q '+glbls.sites[i1])
         t2 = time.time()
         accesstime = t2 - t1
         glbls.acclock.acquire()
         # list full yet?
         if len(glbls.acctimes) < glbls.ww:
            glbls.acctimes.append(accesstime)
         else:
            glbls.acctimes = glbls.acctimes[1:] + [accesstime]
         glbls.acclock.release()
   else:
      while 1:
         rsp = raw_input('monitor: ')
         if rsp == 'mean': print mean(glbls.acctimes)
         elif rsp == 'median': print median(glbls.acctimes)
         elif rsp == 'all': print all(glbls.acctimes)

def mean(x):
   return sum(x)/len(x)

def median(x):
   y = x
   y.sort()
   return y[len(y)/2]  # a little sloppy

def all(x):
   return x

def main():
   nthr = int(sys.argv[3])  # number of threads
   for thr in range(nthr):
      thread.start_new_thread(probe,(thr,))
   while 1: continue

if __name__ == '__main__':
   main()

```

3.1.2 The `threading` Module

The program below treats the same network client/server application considered in Section 3.1.1, but with the more sophisticated **threading** module. The client program stays the same, since it didn't involve threads in the first place. Here is the new server code:

```
# simple illustration of threading module

# multiple clients connect to server; each client repeatedly sends a
# value k, which the server adds to a global string v and echos back
# to the client; k = '' means the client is dropping out; when all
# clients are gone, server prints final value of v

# this is the server

import socket  # networking module
import sys
import threading

# class for threads, subclassed from threading.Thread class
class srvr(threading.Thread):
   # v and vlock are now class variables
   v = ''
   vlock = threading.Lock()
   id = 0  # I want to give an ID number to each thread, starting at 0
   def __init__(self,clntsock):
      # invoke constructor of parent class
      threading.Thread.__init__(self)
      # add instance variables
      self.myid = srvr.id
      srvr.id += 1
      self.myclntsock = clntsock
   # this function is what the thread actually runs; the required name
   #    is run(); threading.Thread.start() calls threading.Thread.run(),
   #    which is always overridden, as we are doing here
   def run(self):
      while 1:
         # receive letter from client, if it is still connected
         k = self.myclntsock.recv(1)
         if k == '': break
         # update v in an atomic manner
         srvr.vlock.acquire()
         srvr.v += k
         srvr.vlock.release()
         # send new v back to client
         self.myclntsock.send(srvr.v)
      self.myclntsock.close()

# set up Internet TCP socket
lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port = int(sys.argv[1])  # server port number
# bind lstn socket to this port
lstn.bind(('', port))
# start listening for contacts from clients (at most 2 at a time)
lstn.listen(5)

nclnt = int(sys.argv[2])  # number of clients

mythreads = []  # list of all the threads
# accept calls from the clients
for i in range(nclnt):
   # wait for call, then get a new socket to use for this client,
   #    and get the client's address/port tuple (though not used)
   (clnt,ap) = lstn.accept()
   # make a new instance of the class srvr
   s = srvr(clnt)
   # keep a list all threads
   mythreads.append(s)
   # threading.Thread.start calls threading.Thread.run(), which we
   #    overrode in our definition of the class srvr
   s.start()

# shut down the server socket, since it's not needed anymore
lstn.close()

# wait for all threads to finish
for s in mythreads:
   s.join()

print 'the final value of v is', srvr.v
```

Again, let's look at the main data structure first:

```
class srvr(threading.Thread):
```

The **threading** module contains a class **Thread**, any instance of which represents one thread. A typical application will subclass this class, for two reasons. First, we will probably have some application-specific variables or methods to be used. Second, the class **Thread** has a member method **run()** which is meant to be overridden, as you will see below.

Consistent with OOP philosophy, we might as well put the old globals in as class variables:

```
v = ''
vlock = threading.Lock()
```

Note that class variable code is executed immediately upon execution of the program, as opposed to when the first object of this class is created. So, the lock is created right away.

```
id = 0
```

This is to set up ID numbers for each of the threads. We don't use them here, but they might be useful in debugging or in future enhancement of the code.

```
def __init__(self,clntsock):
   ...
   self.myclntsock = clntsock

# ``main'' program
...
   (clnt,ap) = lstn.accept()
   s = srvr(clnt)
```

The "main" program, in creating an object of this class for the client, will pass as an argument the socket for that client. We then store it as a member variable for the object.

```
def run(self):
   ...
```

As noted earlier, the **Thread** class contains a member method **run()**. This is a dummy, to be overridden with the application-specific function to be run by the thread. It is invoked by the method **Thread.start()**, called here in the main program. As you can see above, it is pretty much the same as the previous code in Section 3.1.1 which used the **thread** module, adapted to the class environment.

One thing that is quite different in this program is the way we end it:

```
for s in mythreads:
   s.join()
```

The **join()** method in the class **Thread** blocks until the given thread exits. (The threads manager puts the main thread in Sleep state, and when the given thread exits, the manager changes that state to Run.) The overall effect of this loop, then, is that the main program will wait at that point until all the threads are done. They "join" the main program. This is a much cleaner approach than what we used earlier, and it is also more efficient, since the main program will not be given any turns in which it wastes time looping around doing nothing, as in the program in Section 3.1.1 in the line

```
while nclnt > 0: pass
```

Here we maintained our own list of threads. However, we could also get one via the call **threading.enumerate()**. If placed after the **for** loop in our server code above, for instance as

```
print threading.enumerate()
```

we would get output like

```
[<_MainThread(MainThread, started)>, <srvr(Thread-1, started)>,
<srvr(Thread-2, started)>]
```

Here's another example, which finds and counts prime numbers, again not assumed to be efficient:

```
#!/usr/bin/env python

# prime number counter, based on Python threading class

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what's left at the end are the primes

import sys
import math
import threading

class prmfinder(threading.Thread):
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class
   prime = (n+1) * [1]  # 1 means assumed prime, until find otherwise
   nextk = 2  # next value to try crossing out with
   nextklock = threading.Lock()
   def __init__(self,id):
      threading.Thread.__init__(self)
      self.myid = id
   def run(self):
      lim = math.sqrt(prmfinder.n)
      nk = 0  # count of k's done by this thread, to assess load balance
      while 1:
         # find next value to cross out with
         prmfinder.nextklock.acquire()
         k = prmfinder.nextk
         prmfinder.nextk += 1
         prmfinder.nextklock.release()
         if k > lim: break
         nk += 1  # increment workload data
         if prmfinder.prime[k]:  # now cross out
            r = prmfinder.n / k
            for i in range(2,r+1):
               prmfinder.prime[i*k] = 0
      print 'thread', self.myid, 'exiting; processed', nk, 'values of k'

def main():
   for i in range(prmfinder.nthreads):
      pf = prmfinder(i)  # create thread i
      prmfinder.thrdlist.append(pf)
      pf.start()
   for thrd in prmfinder.thrdlist: thrd.join()
   print 'there are', reduce(lambda x,y: x+y, prmfinder.prime) - 2, 'primes'

if __name__ == '__main__':
    main()
```

3.2 Condition Variables

3.2.1 General Ideas

We saw in the last section that **threading.Thread.join()** avoids the need for wasteful looping in **main()**, while the latter is waiting for the other threads to finish. In fact, it is very common in threaded programs to have situations in which one thread needs to wait for something to occur in another thread. Again, in such situations we would not want the waiting thread to engage in wasteful looping.

The solution to this problem is **condition variables**. As the name implies, these are variables used by code to wait for a certain condition to occur. Most threads systems include provisions for these, and Python's **threading** package is no exception.

The **pthreads** package, for instance, has a type **pthread_cond** for such variables, and has functions such as **pthread_cond_wait()**, which a thread calls to wait for an event to occur, and **pthread_cond_signal()**, which another thread calls to announce that the event now has occurred.

But as is typical with Python in so many things, it is easier for us to use condition variables in Python than in C. At the first level, there is the class **threading.Condition**, which corresponds well to the condition variables available in most threads systems. However, at this level condition variables are rather cumbersome to use, as not only do we need to set up condition variables but we also need to set up extra locks to guard them. This is necessary in any threading system, but it is a nuisance to deal with.

So, Python offers a higher-level class, **threading.Event**, which is just a wrapper for **threading.Condition**, but which does all the condition lock operations behind the scenes, alleviating the programmer of having to do this work.

3.2.2 `Event` Example

Following is an example of the use of **threading.Event**. It searches a given network host for servers at various ports on that host. (This is called a **port scanner**.) As noted in a comment, the threaded operation used here would make more sense if many hosts were to be scanned, rather than just one, as each **connect()** operation does take some time. But even on the same machine, if a server is active but busy enough that we never get to connect to it, it may take a long for the attempt to timeout. It is common to set up Web operations to be threaded for that reason. We could also have each thread check a block of ports on a host, not just one, for better efficiency.

The use of threads is aimed at checking many ports in parallel, one per thread. The program has a self-imposed limit on the number of threads. If **main()** is ready to start checking another port but we are at the thread limit, the code in **main()** waits for the number of threads to drop below the limit. This is accomplished by a condition wait, implemented through the **threading.Event** class.

```
# portscanner.py:  checks for active ports on a given machine; would be
# more realistic if checked several hosts at once; different threads
# check different ports; there is a self-imposed limit on the number of
# threads, and the event mechanism is used to wait if that limit is
# reached

# usage:  python portscanner.py host maxthreads

```

```
import sys, threading, socket

class scanner(threading.Thread):
   tlist = []  # list of all current scanner threads
   maxthreads = int(sys.argv[2])  # max number of threads we’re allowing
   evnt = threading.Event()  # event to signal OK to create more threads
   lck =  threading.Lock()  # lock to guard tlist
   def __init__(self,tn,host):
      threading.Thread.__init__(self)
      self.threadnum = tn  # thread ID/port number
      self.host = host  # checking ports on this host
   def run(self):
      s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
      try:
         s.connect((self.host, self.threadnum))
         print "%d:  successfully connected" % self.threadnum
         s.close()
      except:
         print "%d:  connection failed" % self.threadnum
      # thread is about to exit; remove from list, and signal OK if we
      # had been up against the limit
      scanner.lck.acquire()
      scanner.tlist.remove(self)
      print "%d:  now active --" % self.threadnum, scanner.tlist
      if len(scanner.tlist) == scanner.maxthreads-1:
         scanner.evnt.set()
         scanner.evnt.clear()
      scanner.lck.release()
   def newthread(pn,hst):
      scanner.lck.acquire()
      sc = scanner(pn,hst)
      scanner.tlist.append(sc)
      scanner.lck.release()
      sc.start()
      print "%d:  starting check" % pn
      print "%d:  now active --" % pn, scanner.tlist
   newthread = staticmethod(newthread)

def main():
   host = sys.argv[1]
   for i in range(1,100):
      scanner.lck.acquire()
      print "%d:  attempting check" % i
      # check to see if we’re at the limit before starting a new thread
      if len(scanner.tlist) >= scanner.maxthreads:
         # too bad, need to wait until not at thread limit
         print "%d:  need to wait" % i
         scanner.lck.release()
         scanner.evnt.wait()
      else:
         scanner.lck.release()
      scanner.newthread(i,host)
   for sc in scanner.tlist:
      sc.join()

if __name__ == ’__main__’:
    main()
```

As you can see, when **main()** discovers that we are at our self-imposed limit of number of active threads, we back off by calling **threading.Event.wait()**. At that point **main()**—which, recall, is also a thread—blocks. It will not be given any more timeslices for the time being. When some active thread exits, we have it call **threading.Event.set()** and **threading.Event.clear()**. The threads manager reacts to the former by moving all threads which had been waiting for this event—in our case here, only **main()**—from Sleep state to Run state; **main()** will eventually get another timeslice.

The call to **threading.Event.clear()** is crucial. The word *clear* here means that **threading.Event.clear()** is clearing the occurence of the event. Without this, any subsequent call to **threading.Event.wait()** would immediately return, even though the condition has not been met yet.

Note carefully the use of locks. The **main()** thread adds items to **tlist**, while the other threads delete items (delete themselves, actually) from it. These operations must be atomic, and thus must be guarded by locks.

I've put in a lot of extra **print** statements so that you can get an idea as to how the threads' execution is interleaved. Try running the program.[8] But remember, the program may appear to hang for a long time if a server is active but so busy that the attempt to connect times out.

3.2.3 Other `threading` Classes

The function **Event.set()** "wakes" all threads that are waiting for the given event. That didn't matter in our example above, since only one thread (**main()**) would ever be waiting at a time in that example. But in more general applications, we sometimes want to wake only one thread instead of all of them. For this, we can revert to working at the level of **threading.Condition** instead of **threading.Event**. There we have a choice between using **notify()** or **notifyAll()**.

The latter is actually what is called internally by **Event.set()**. But **notify()** instructs the threads manager to wake just one of the waiting threads (we don't know which one).

The class **threading.Semaphore** offers semaphore operations. Other classes of advanced interest are **threading.RLock** and **threading.Timer**.

3.3 Threads Internals

The thread manager acts like a "mini-operating system." Just like a real OS maintains a table of processes, a thread system's thread manager maintains a table of threads. When one thread gives up the CPU, or has its turn pre-empted (see below), the thread manager looks in the table for another thread to activate. Whichever thread is activated will then resume execution where it had left off, i.e. where its last turn ended.

[8]Disclaimer: Not guaranteed to be bug-free.

Just as a process is either in Run state or Sleep state, the same is true for a thread. A thread is either ready to be given a turn to run, or is waiting for some event. The thread manager will keep track of these states, decide which thread to run when another has lost its turn, etc.

3.3.1 Kernel-Level Thread Managers

Here each thread really is a process, and for example will show up on Unix systems when one runs the appropriate **ps** process-list command, say **ps axH**. The threads manager is then the OS.

The different threads set up by a given application program take turns running, among all the other processes.

This kind of thread system is is used in the Unix **pthreads** system, as well as in Windows threads.

3.3.2 User-Level Thread Managers

User-level thread systems are "private" to the application. Running the **ps** command on a Unix system will show only the original application running, not all the threads it creates. Here the threads are not pre-empted; on the contrary, a given thread will continue to run until it voluntarily gives up control of the CPU, either by calling some "yield" function or by calling a function by which it requests a wait for some event to occur.[9]

A typical example of a user-level thread system is **pth**.

3.3.3 Comparison

Kernel-level threads have the advantage that they can be used on multiprocessor systems, thus achieving true parallelism between threads. This is a major advantage.

On the other hand, in my opinion user-level threads also have a major advantage in that they allow one to produce code which is much easier to write, is easier to debug, and is cleaner and clearer. This in turn stems from the non-preemptive nature of user-level threads; application programs written in this manner typically are not cluttered up with lots of lock/unlock calls (details on these below), which are needed in the pre-emptive case.

3.3.4 The Python Thread Manager

Python "piggybacks" on top of the OS' underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the **ps** output.

[9]In typical user-level thread systems, an external event, such as an I/O operation or a signal, will also also cause the current thread to relinquish the CPU.

However, Python's thread manager imposes further structure on top of the OS threads. It keeps track of how long a thread has been executing, in terms of the number of Python **byte code** instructions that have executed.[10] When that reaches a certain number, by default 100, the thread's turn ends. In other words, the turn can be pre-empted either by the hardware timer and the OS, or when the interpreter sees that the thread has executed 100 byte code instructions.[11]

3.3.4.1 The GIL

In the case of CPython (but not Jython or Iron Python) Most importantly, there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that only one thread runs at a time, in order to facilitate easy garbage collection.

Suppose we have a C program with three threads, which I'll call X, Y and Z. Say currently Y is running. After 30 milliseconds (or whatever the quantum size has been set to by the OS), Y will be interrupted by the timer, and the OS will start some other process. Say the latter, which I'll call Q, is a different, unrelated program. Eventually Q's turn will end too, and let's say that the OS then gives X a turn. From the point of view of our X/Y/Z program, i.e. ignoring Q, control has passed from Y to X. The key point is that the point within Y at which that event occurs is random (with respect to where Y is at the time), based on the time of the hardware interrupt.

By contrast, say my Python program has three threads, U, V and W. Say V is running. The hardware timer will go off at a random time, and again Q might be given a turn, *but* definitely neither U nor W will be given a turn, because the Python interpreter had earlier made a call to the OS which makes U and W wait for the GIL to become unlocked.

Let's look at this a little closer. The key point to note is that the Python interpreter itself is threaded, say using **pthreads**. For instance, in our X/Y/Z example above, when you ran **ps axH**, you would see three Python processes/threads. I just tried that on my program **thsvr.py**, which creates two threads, with a command-line argument of 2000 for that program. Here is the relevant portion of the output of **ps axH**:

```
28145 pts/5    Rl     0:09 python thsvr.py 2000
28145 pts/5    Sl     0:00 python thsvr.py 2000
28145 pts/5    Sl     0:00 python thsvr.py 2000
```

What has happened is the Python interpreter has spawned two child threads, one for each of my threads in **thsvr.py**, in addition to the interpreter's original thread, which runs my **main()**. Let's call those threads UP, VP and WP. Again, these are the threads that the OS sees, while U, V and W are the threads that I see—or think I see, since they are just virtual.

[10]This is the "machine language" for the Python virtual machine.
[11]The author thanks Alex Martelli for a helpful clarification.

The GIL is a **pthreads** lock. Say V is now running. Again, what that actually means on my real machine is that VP is running. VP keeps track of how long V has been executing, in terms of the number of Python **byte code** instructions that have executed. When that reaches a certain number, by default 100, UP will release the GIL by calling **pthread_mutex_unlock()** or something similar.

The OS then says, “Oh, were any threads waiting for that lock?” It then basically gives a turn to UP or WP (we can’t predict which), which then means that from my point of view U or W starts, say U. Then VP and WP are still in Sleep state, and thus so are my V and W.

So you can see that it is the Python interpreter, not the hardware timer, that is determining how long a thread’s turn runs, relative to the other threads in my program. Again, Q might run too, but within this Python program there will be no control passing from V to U or W simply because the timer went off; such a control change will only occur when the Python interpreter wants it to. This will be either after the 100 byte code instructions or when U reaches an I/O operation or other wait-event operation.

So, the bottom line is that while Python uses the underlying OS threads system as its base, it superimposes further structure in terms of transfer of control between threads.

3.3.4.2 Implications for Randomness and Need for Locks

I mentioned in Section 3.3.2 that non-pre-emptive threading is nice because one can avoid the code clutter of locking and unlocking (details of lock/unlock below). Since, barring I/O issues, a thread working on the same data would seem to always yield control at exactly the same point (i.e. at 100 byte code instruction boundaries), Python would seem to be deterministic and non-pre-emptive. However, it will not quite be so simple.

First of all, there is the issue of I/O, which adds randomness. There may also be randomness in how the OS chooses the first thread to be run, which could affect computation order and so on.

Finally, there is the question of atomicity in Python operations: The interpreter will treat any Python virtual machine instruction as indivisible, thus not needing locks in that case. But the bottom line will be that unless you know the virtual machine well, you should use locks at all times.

3.4 The `multiprocessing` Module

CPython’s GIL is the subject of much controversy. As we saw in Section 3.3.4.1, it prevents running true parallel applications when using the **thread** or **threading** modules.

That might not seem to be too severe a restriction—after all if you really need the speed, you probably won’t use a scripting language in the first place. But a number of people took the point of view that, given that they have decided to use Python no matter what, they would like to get the best speed subject to that restriction.

So, there was much grumbling about the GIL.

Thus, later the **multiprocessing** module was developed, which enables true parallel processing with Python on a multiprocore machine, with an interface very close to that of the **threading** module.

Moreover, one can run a program across machines! In other words, the **multiprocessing** module allows to run several threads not only on the different cores of one machine, but on many machines at once, in cooperation in the same manner that threads cooperate on one machine. By the way, this idea is similar to something I did for Perl some years ago (PerlDSM: A Distributed Shared Memory System for Perl. *Proceedings of PDPTA 2002*, 63-68). We will not cover the cross-machine case here.

So, let's go to our first example, a simulation application that will find the probability of getting a total of exactly k dots when we roll n dice:

```
# dice probability finder, based on Python multiprocessing class

# usage:  python DiceProb.py n k nreps nthreads
#   where we wish to find the probability of getting a total of k dots
#   when we roll n dice; we'll use nreps total repetitions of the
#   simulation, dividing those repetitions among nthreads threads

import sys
import random
from multiprocessing import Process, Lock, Value

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   k = int(sys.argv[2])
   nreps = int(sys.argv[3])
   nthreads = int(sys.argv[4])
   thrdlist = []  # list of all instances of this class

def worker(id,tot,totlock):
   mynreps = glbls.nreps/glbls.nthreads
   r = random.Random()  # set up random number generator
   count = 0  # number of times get total of k
   for i in range(mynreps):
      if rolldice(r) == glbls.k:
         count += 1
   totlock.acquire()
   tot.value += count
   totlock.release()
   # check for load balance
   print 'thread', id, 'exiting; total was', count

def rolldice(r):
   ndots = 0
   for roll in range(glbls.n):
      dots = r.randint(1,6)
      ndots += dots
   return ndots

def main():
   tot = Value('i',0)
   totlock = Lock()
   for i in range(glbls.nthreads):
      pr = Process(target=worker, args=(i,tot,totlock))
      glbls.thrdlist.append(pr)
      pr.start()
   for thrd in glbls.thrdlist: thrd.join()
   # adjust for truncation, in case nthreads doesn't divide nreps evenly
   actualnreps = glbls.nreps/glbls.nthreads * glbls.nthreads
   print 'the probability is',float(tot.value)/actualnreps

if __name__ == '__main__':
    main()
```

As in any simulation, the longer one runs it, the better the accuracy is likely to be. Here we run the simulation **nreps** times, but divide those repetitions among the threads. This is an example of an "embarrassingly parallel" application, so we should get a good speedup (not shown here).

So, how does it work? The general structure looks similar to that of the Python **threading** module, using **Process()** to create a create a thread, **start()** to get it running, **Lock()** to create a lock, **acquire()** and **release()** to lock and unlock a lock, and so on.

The main difference, though, is that globals are not automatically shared. Instead, shared variables must be created using **Value** for a scalar and **Array** for an array. Unlike Python in general, here one must specify a data type, 'i' for integer and 'd' for double (floating-point). (One can use **Namespace** to create more complex types, at some cost in performance.) One also specifies the initial value of the variable. One must pass these variables explicitly to the functions to be run by the threads, in our case above the function **worker()**. Note carefully that the shared variables are still accessed syntactically as if they were globals.

Here's the prime number-finding program from before, now using **multiprocessing**:

```
#!/usr/bin/env python

# prime number counter, based on Python multiprocessing class

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what's left at the end are the primes

import sys
import math
from multiprocessing import Process, Lock, Array, Value

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class

```

```
def prmfinder(id,prm,nxtk,nxtklock):
   lim = math.sqrt(glbls.n)
   nk = 0  # count of k's done by this thread, to assess load balance
   while 1:
      # find next value to cross out with
      nxtklock.acquire()
      k = nxtk.value
      nxtk.value = nxtk.value + 1
      nxtklock.release()
      if k > lim: break
      nk += 1  # increment workload data
      if prm[k]:  # now cross out
         r = glbls.n / k
         for i in range(2,r+1):
            prm[i*k] = 0
   print 'thread', id, 'exiting; processed', nk, 'values of k'

def main():
   prime = Array('i',(glbls.n+1) * [1])  # 1 means prime, until find otherwise
   nextk = Value('i',2)  # next value to try crossing out with
   nextklock = Lock()
   for i in range(glbls.nthreads):
      pf = Process(target=prmfinder, args=(i,prime,nextk,nextklock))
      glbls.thrdlist.append(pf)
      pf.start()
   for thrd in glbls.thrdlist: thrd.join()
   print 'there are', reduce(lambda x,y: x+y, prime) - 2, 'primes'

if __name__ == '__main__':
    main()
```

The main new item in this example is use of **Array()**.

One can use the **Pool** class to create a set of threads, rather than doing so "by hand" in a loop as above. You can start them with various initial values for the threads using **Pool.map()**, which works similarly to Python's ordinary **map()** function.

The **multiprocessing** documentation warns that shared items may be costly, and suggests using **Queue** and **Pipe** where possible. We will cover the former in the next section. Note, though, that in general it's difficult to get much speedup (or difficult even to avoid slowdown!) with non-"embarrassingly parallel" applications.

3.5 The Queue Module for Threads and Multiprocessing

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

Clearly one needs to guard the queue with locks. But Python provides the **Queue** module to take care of all the lock creation, locking and unlocking, and so on. This means we don't have to bother with it, and the code will probably be faster.

Queue is implemented for both **threading** and **multiprocessing**, in almost identical forms. This is good, because the documentation for **multiprocessing** is rather sketchy, so you can turn to the docs for `threading` for more details.

The function **put()** in Queue adds an element to the end of the queue, while **get()** will remove the head of the queue, again without the programmer having to worry about race conditions.

Note that **get()** will block if the queue is currently empty. An alternative is to call it with **block=False**, within a **try/except** construct. One can also set timeout periods.

Here once again is the prime number example, this time done with **Queue**:

```
#!/usr/bin/env python

# prime number counter, based on Python multiprocessing class with
# Queue

# usage:  python PrimeThreading.py n nthreads
#   where we wish the count of the number of primes from 2 to n, and to
#   use nthreads to do the work

# uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
# cross out all the multiples of 2, then of 3, then of 5, etc., up to
# sqrt(n); what's left at the end are the primes

import sys
import math
from multiprocessing import Process, Array, Queue

class glbls:  # globals, other than shared
   n = int(sys.argv[1])
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class

def prmfinder(id,prm,nxtk):
   nk = 0  # count of k's done by this thread, to assess load balance
   while 1:
      # find next value to cross out with
      try: k = nxtk.get(False)
      except: break
      nk += 1  # increment workload data
      if prm[k]:  # now cross out
         r = glbls.n / k
         for i in range(2,r+1):
            prm[i*k] = 0
   print 'thread', id, 'exiting; processed', nk, 'values of k'

def main():
   prime = Array('i',(glbls.n+1) * [1])  # 1 means prime, until find otherwise
   nextk = Queue()  # next value to try crossing out with
   lim = int(math.sqrt(glbls.n)) + 1  # fill the queue with 2...sqrt(n)
   for i in range(2,lim): nextk.put(i)
   for i in range(glbls.nthreads):
      pf = Process(target=prmfinder, args=(i,prime,nextk))
      glbls.thrdlist.append(pf)
      pfs.append(pf)
      pf.start()
   for thrd in glbls.thrdlist: thrd.join()
   print 'there are', reduce(lambda x,y: x+y, prime) - 2, 'primes'

if __name__ == '__main__':
    main()
```

The way **Queue** is used here is to put all the possible "crosser-outers," obtained in the variable **nextk** in the previous versions of this code, into a queue at the outset. One then uses `get()` to pick up work from the queue. Look Ma, no locks!

Below is an example of queues in an in-place quicksort. (Again, the reader is warned that this is just an example, not claimed to be efficient.)

The work items in the queue are a bit more involved here. They have the form **(i,j,k)**, with the first two elements of this tuple meaning that the given array chunk corresponds to indices **i** through **j** of **x**, the original array to be sorted. In other words, whichever thread picks up this chunk of work will have the responsibility of handling that particular section of **x**.

Quicksort, of course, works by repeatedly splitting the original array into smaller and more numerous chunks. Here a thread will split its chunk, taking the lower half for itself to sort, but placing the upper half into the queue, to be available for other chunks that have not been assigned any work yet. I've written the algorithm so that as soon as all threads have gotten some work to do, no more splitting will occur. That's where the value of **k** comes in. It tells us the split number of this chunk. If it's equal to **nthreads-1**, this thread won't split the chunk.

```
# Quicksort and test code, based on Python multiprocessing class and
# Queue

# code is incomplete, as some special cases such as empty subarrays
# need to be accounted for

# usage:  python QSort.py n nthreads
#   where we wish to test the sort on a random list of n items,
#   using nthreads to do the work

import sys
import random
from multiprocessing import Process, Array, Queue

class glbls:  # globals, other than shared
   nthreads = int(sys.argv[2])
   thrdlist = []  # list of all instances of this class
   r = random.Random(9876543)

def sortworker(id,x,q):
   chunkinfo = q.get()
   i = chunkinfo[0]
   j = chunkinfo[1]
   k = chunkinfo[2]
   if k < glbls.nthreads - 1:  # need more splitting?
      splitpt = separate(x,i,j)
      q.put((splitpt+1,j,k+1))
      # now, what do I sort?
      rightend = splitpt + 1
   else: rightend = j
   tmp = x[i:(rightend+1)]  # need copy, as Array type has no sort() method
   tmp.sort()
   x[i:(rightend+1)] = tmp

def separate(xc, low, high):  # common algorithm; see Wikipedia
   pivot = xc[low]  # would be better to take, e.g., median of 1st 3 elts
   (xc[low],xc[high]) = (xc[high],xc[low])
   last = low
   for i in range(low,high):
      if xc[i] <= pivot:
         (xc[last],xc[i]) = (xc[i],xc[last])
         last += 1
   (xc[last],xc[high]) = (xc[high],xc[last])
   return last

def main():
   tmp = []
   n = int(sys.argv[1])
   for i in range(n): tmp.append(glbls.r.uniform(0,1))
   x = Array('d',tmp)
   # work items have form (i,j,k), meaning that the given array chunk
   # corresponds to indices i through j of x, and that this is the kth
   # chunk that has been created, x being the 0th
   q = Queue()  # work queue
   q.put((0,n-1,0))
   for i in range(glbls.nthreads):
      p = Process(target=sortworker, args=(i,x,q))
      glbls.thrdlist.append(p)
      p.start()
   for thrd in glbls.thrdlist: thrd.join()
   if n < 25: print x[:]

if __name__ == '__main__':
    main()
```

3.6 Debugging Threaded and Multiprocessing Python Programs

Debugging is always tough with parallel programs, including threads programs. It's especially difficult with pre-emptive threads; those accustomed to debugging non-threads programs find it rather jarring to see sudden changes of context while single-stepping through code. Tracking down the cause of deadlocks can be very hard. (Often just getting a threads program to end properly is a challenge.)

Another problem which sometimes occurs is that if you issue a "next" command in your debugging tool, you may end up inside the internal threads code. In such cases, use a "continue" command or something like that to extricate yourself.

Unfortunately, as of April 2010, I know of no debugging tool that works with **multiprocessing**. However, one can do well with **thread** and **threading**.

3.6.1 Using PDB to Debug Threaded Programs

Using PDB is a bit more complex when threads are involved. One cannot, for instance, simply do something like this:

```
pdb.py buggyprog.py
```

because the child threads will not inherit the PDB process from the main thread. You can still run PDB in the latter, but will not be able to set breakpoints in threads.

What you can do, though, is invoke PDB from *within* the function which is run by the thread, by calling **pdb.set_trace()** at one or more points within the code:

```
import pdb
pdb.set_trace()
```

In essence, those become breakpoints.

For example, in our program **srvr.py** in Section 3.1.1, we could add a PDB call at the beginning of the loop in **serveclient()**:

```
while 1:
   import pdb
   pdb.set_trace()
   # receive letter from client, if it is still connected
   k = c.recv(1)
   if k == '': break
```

You then run the program directly through the Python interpreter as usual, NOT through PDB, but then the program suddenly moves into debugging mode on its own. At that point, one can then step through the code using the **n** or **s** commands, query the values of variables, etc.

PDB's **c** ("continue") command still works. Can one still use the **b** command to set additional breakpoints? Yes, but it might be only on a one-time basis, depending on the context. A breakpoint might work only once, due to a scope problem. Leaving the scope where we invoked PDB causes removal of the trace object. Thus I suggested setting up the trace inside the loop above.

Of course, you can get fancier, e.g. setting up "conditional breakpoints," something like:

```
debugflag = int(sys.argv[1])
...
if debugflag == 1:
   import pdb
   pdb.set_trace()
```

Then, the debugger would run only if you asked for it on the command line. Or, you could have multiple **debugflag** variables, for activating/deactivating breakpoints at various places in the code.

Moreover, once you get the `(Pdb)` prompt, you could set/reset those flags, thus also activating/deactivating breakpoints.

Note that local variables which were set before invoking PDB, including parameters, are not accessible to PDB.

Make sure to insert code to maintain an ID number for each thread. This really helps when debugging.

3.6.2 RPDB2 and Winpdb

The Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[12] is very good. Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't. Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

Another very promising debugger that handles threads is PYDB, by Rocky Bernstein (not to be confused with an earlier debugger of the same name). You can obtain it from `http://code.google.com/p/pydbgr/` or the older version at `http://bashdb.sourceforge.net/pydb/`. Invoke it on your code **x.py** by typing

```
$ pydb --threading x.py your_command_line_args_for_x
```

[12]No, it's not just for Microsoft Windows machines, in spite of the name.

Chapter 4

Introduction to OpenMP

OpenMP has become the *de facto* standard for shared-memory programming.

4.1 Overview

OpenMP has become the environment of choice for many, if not most, practitioners of shared-memory parallel programming. It consists of a set of directives which are added to one's C/C++/FORTRAN code that manipulate threads, without the programmer him/herself having to deal with the threads directly. This way we get "the best of both worlds"—the true parallelism of (nonpreemptive) threads and the pleasure of avoiding the annoyances of threads programming.

Most OpenMP constructs are expressed via **pragmas**, i.e. directives. The syntax is

```
#pragma omp ......
```

The number sign must be the first nonblank character in the line.

4.2 Running Example

The following example, implementing Dijkstra's shortest-path graph algorithm, will be used throughout this tutorial, with various OpenMP constructs being illustrated later by modifying this code:

```
// Dijkstra.c

// OpenMP example program:  Dijkstra shortest-path finder in a
// bidirectional graph; finds the shortest path from vertex 0 to all
// others

// usage:  dijkstra nv print

// where nv is the size of the graph, and print is 1 if graph and min
// distances are to be printed out, 0 otherwise

#include <omp.h>

// global variables, shared by all threads by default

int nv,  // number of vertices
    *notdone, // vertices not checked yet
    nth,  // number of threads
    chunk,  // number of vertices handled by each thread
    md,  // current min over all threads
    mv,  // vertex which achieves that min
    largeint = -1;  // max possible unsigned int

unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
         // ohd[i*nv+j]
         *mind;  // min distances found so far

void init(int ac, char **av)
{  int i,j,tmp;
   nv = atoi(av[1]);
   ohd = malloc(nv*nv*sizeof(int));
   mind = malloc(nv*sizeof(int));
   notdone = malloc(nv*sizeof(int));
   // random graph
   for (i = 0; i < nv; i++)
      for (j = i; j < nv; j++)  {
         if (j == i) ohd[i*nv+i] = 0;
         else  {
            ohd[nv*i+j] = rand() % 20;
            ohd[nv*j+i] = ohd[nv*i+j];
         }
      }
   for (i = 1; i < nv; i++)  {
      notdone[i] = 1;
      mind[i] = ohd[i];
   }
}

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

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

void dowork()
{
   #pragma omp parallel
   {  int startv,endv,  // start, end vertices for my thread
          step,  // whole procedure goes nv steps
          mymv,  // vertex which attains the min value in my chunk
          me = omp_get_thread_num();
          unsigned mymd;  // min value found by this thread
      #pragma omp single
      {  nth = omp_get_num_threads();
         if (nv % nth != 0) {
            printf("nv must be divisible by nth\n");
            exit(1);
         }
         chunk = nv/nth;
         printf("there are %d threads\n",nth);
      }
      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
         {  if (mymd < md)
               {  md = mymd; mv = mymv;  }
         }
         #pragma omp barrier
         // mark new vertex as done
         #pragma omp single
         {  notdone[mv] = 0;  }
         // now update my section of mind
         updatemind(startv,endv);
         #pragma omp barrier
      }
   }
}

int main(int argc, char **argv)
{  int i,j,print;
   double startime,endtime;
   init(argc,argv);
   startime = omp_get_wtime();
   // parallel
   dowork();
   // back to single thread
   endtime = omp_get_wtime();
   printf("elapsed time:  %f\n",endtime-startime);
   print = atoi(argv[2]);
   if (print)  {
      printf("graph weights:\n");
      for (i = 0; i < nv; i++)  {
         for (j = 0; j < nv; j++)
            printf("%u  ",ohd[nv*i+j]);
         printf("\n");
      }
      printf("minimum distances:\n");
      for (i = 1; i < nv; i++)
         printf("%u\n",mind[i]);
   }
}
```

The constructs will be presented in the following sections, but first the algorithm will be explained.

4.2.1 The Algorithm

The code implements the Dijkstra algorithm for finding the shortest paths from vertex 0 to the other vertices in an N-vertex undirected graph. Pseudocode for the algorithm is shown below, with the array G assumed to contain the one-hop distances from 0 to the other vertices.

```
Done = {0}  # vertices checked so far
NewDone = None  # currently checked vertex
NonDone = {1,2,...,N-1}  # vertices not checked yet
for J = 0 to N-1 Dist[J] = G(0,J)  # initialize shortest-path lengths

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
         # check if there is a shorter path from 0 to K through NewDone
         # than our best so far
         Dist[K] = min(Dist[K],Dist[NewDone]+G[NewDone,K])
```

At each iteration, the algorithm finds the closest vertex J to 0 among all those not yet processed, and then updates the list of minimum distances to each vertex from 0 by considering paths that go through J. Two obvious potential candidate part of the algorithm for parallelization are the “find J” and “for K” lines, and the above OpenMP code takes this approach.

4.2.2 The OpenMP `parallel` Pragma

As can be seen in the comments in the lines

```
   // parallel
   dowork();
   // back to single thread
```

the function **main()** is run by a **master thread**, which will then branch off into many threads running **dowork()** in parallel. The latter feat is accomplished by the directive in the lines

```
void dowork()
{
   #pragma omp parallel
   {  int startv,endv,  // start, end vertices for this thread
          step,  // whole procedure goes nv steps
          mymv,  // vertex which attains that value
          me = omp_get_thread_num();
```

That directive sets up a team of threads (which includes the master), all of which execute the block following the directive in parallel.[1] Note that, unlike the **for** directive which will be discussed below, the **parallel** directive leaves it up to the programmer as to how to partition the work. In our case here, we do that by setting the range of vertices which this thread will process:

```
      startv = me * chunk;
      endv = startv + chunk - 1;
```

Again, keep in mind that *all* of the threads execute this code, but we've set things up with the variable **me** so that different threads will work on different vertices. This is due to the OpenMP call

```
          me = omp_get_thread_num();
```

which sets **me** to the thread number for this thread.

4.2.3 Scope Issues

Note carefully that in

```
   #pragma omp parallel
   {  int startv,endv,  // start, end vertices for this thread
          step,  // whole procedure goes nv steps
          mymv,  // vertex which attains that value
          me = omp_get_thread_num();
```

[1]There is an issue here of thread startup time. The OMPi compiler sets up threads at the outset, so that that startup time is incurred only once. When a **parallel** construct is encountered, they are awakened. At the end of the construct, they are suspended again, until the next **parallel** construct is reached.

the pragma comes *before* the declaration of the local variables. That means that all of them are "local" to each thread, i.e. not shared by them. But if a work sharing directive comes within a function but *after* declaration of local variables, those variables are actually "global" to the code in the directive, i.e. they *are* shared in common among the threads.

This is the default, but you can change these properties, e.g. using the **shared** keyword. For instance,

```
#pragma omp parallel private(x,y)
```

would make **x** and **y** nonshared even if they were declared above the directive line.

It is crucial to keep in mind that variables which are global to the program (in the C/C++ sense) are automatically global to all threads. This is the primary means by which the threads communicate with each other.

4.2.4 The OpenMP `single` Pragma

In some cases we want just one thread to execute some code, even though that code is part of a **parallel** or other **work sharing** block.[2] We use the **single** directive to do this, e.g.:

```
#pragma omp single
{  nth = omp_get_num_threads();
   if (nv % nth != 0) {
      printf("nv must be divisible by nth\n");
      exit(1);
   }
   chunk = nv/nth;
   printf("there are %d threads\n",nth);  }
```

Since the variables **nth** and **chunk** are global and thus shared, we need not have all threads set them, hence our use of **single**.

4.2.5 The OpenMP `barrier` Pragma

As see in the example above, the **barrier** implements a standard barrier, applying to all threads.

4.2.6 Implicit Barriers

Note that there is an implicit barrier at the end of each **single** block, which is also the case for **parallel**, **for**, and **sections** blocks. This can be overridden via the **nowait** clause, e.g.

[2]This is an OpenMP term. The **for** directive is another example of it. More on this below.

```
#pragma omp for nowait
```

Needless to say, the latter should be used with care, and in most cases will not be usable. On the other hand, putting in a barrier where it is not needed would severely reduce performance.

4.2.7 The OpenMP `critical` Pragma

The last construct used in this example is **critical**, for critical sections.

```
#pragma omp critical
{  if (mymd < md)
      {  md = mymd; mv = mymv;  }
}
```

It means what it says, allowing entry of only one thread at a time while others wait. Here we are updating global variables **md** and **mv**, which has to be done atomically, and **critical** takes care of that for us. This is much more convenient than setting up lock variables, etc., which we would do if we were programming threads code directly.

4.3 The OpenMP `for` Pragma

This one breaks up a C/C++ **for** loop, assigning various iterations to various threads. This way the iterations are done in parallel. Of course, that means that they need to be independent iterations, i.e. one iteration cannot depend on the result of another.

4.3.1 Basic Example

Here's how we could use this construct in the Dijkstra program :

```
// Dijkstra.c

// OpenMP example program (OMPi version):  Dijkstra shortest-path finder
// in a bidirectional graph; finds the shortest path from vertex 0 to
// all others

// usage:  dijkstra nv print

// where nv is the size of the graph, and print is 1 if graph and min
// distances are to be printed out, 0 otherwise

#include <omp.h>
```

```

// global variables, shared by all threads by default

int nv,  // number of vertices
    *notdone, // vertices not checked yet
    nth,  // number of threads
    chunk,  // number of vertices handled by each thread
    md,  // current min over all threads
    mv,  // vertex which achieves that min
    largeint = -1;  // max possible unsigned int

unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
                // ohd[i*nv+j]
         *mind;  // min distances found so far

void init(int ac, char **av)
{  int i,j,tmp;
   nv = atoi(av[1]);
   ohd = malloc(nv*nv*sizeof(int));
   mind = malloc(nv*sizeof(int));
   notdone = malloc(nv*sizeof(int));
   // random graph
   for (i = 0; i < nv; i++)
      for (j = i; j < nv; j++)  {
         if (j == i) ohd[i*nv+i] = 0;
         else  {
            ohd[nv*i+j] = rand() % 20;
            ohd[nv*j+i] = ohd[nv*i+j];
         }
      }
   for (i = 1; i < nv; i++)  {
      notdone[i] = 1;
      mind[i] = ohd[i];
   }
}

void dowork()
{
   #pragma omp parallel
   {  int step,  // whole procedure goes nv steps
          mymv,  // vertex which attains that value
          me = omp_get_thread_num(),
          i;
      unsigned mymd;  // min value found by this thread
      #pragma omp single
      {  nth = omp_get_num_threads();
         printf("there are %d threads\n",nth);  }
      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;  }
         mymd = largeint;
         #pragma omp for nowait
         for (i = 1; i < nv; i++)  {
            if (notdone[i] && mind[i] < mymd)  {
               mymd = ohd[i];
               mymv = i;
            }
         }
         // update overall min if mine is smaller
         #pragma omp critical
         {  if (mymd < md)
               {  md = mymd; mv = mymv;  }
         }
         // mark new vertex as done
         #pragma omp single
         {  notdone[mv] = 0;  }
         // now update ohd
         #pragma omp for
         for (i = 1; i < nv; i++)
            if (mind[mv] + ohd[mv*nv+i] < mind[i])
               mind[i] = mind[mv] + ohd[mv*nv+i];
      }
   }
}

int main(int argc, char **argv)
{  int i,j,print;
   init(argc,argv);
   // parallel
   dowork();
   // back to single thread
   print = atoi(argv[2]);
   if (print)  {
      printf("graph weights:\n");
      for (i = 0; i < nv; i++)  {
         for (j = 0; j < nv; j++)
            printf("%u  ",ohd[nv*i+j]);
         printf("\n");
      }
      printf("minimum distances:\n");
      for (i = 1; i < nv; i++)
         printf("%u\n",mind[i]);
   }
}
```

The work which used to be done in the function **findmymin()** is now done here:

```
         #pragma omp for
         for (i = 1; i < nv; i++)  {
            if (notdone[i] && mind[i] < mymd)  {
               mymd = ohd[i];
               mymv = i;
            }
         }
```

Each thread executes one or more of the iterations, i.e. takes responsibility for one or more values of *i*. This occurs in parallel, so as mentioned earlier, the programmer must make sure that the iterations are independent; there is no predicting which threads will do which values of **i**, in which order. By the way, for obvious reasons OpenMP treats the loop index, **i** here, as private even if by context it would be shared.

4.3.2 Nested Loops

If we use the **for** pragma to nested loops, by default the pragma applies only to the outer loop. We can of course insert another **for** pragma inside, to parallelize the inner loop.

Or, starting with OpenMP version 3.0, one can use the **collapse** clause, e.g.

```
#pragma omp parallel for collapse(2)
```

to specify two levels of nesting in the assignment of threads to tasks.

4.3.3 Controlling the Partitioning of Work to Threads

In this default version of the **for** construct, iterations are executed by threads *in unpredictable order*, with each thread taking on *one iteration's worth of work at a time*. Both of these can be changed by the programmer, using the **schedule** clause.

For instance, our original version of our program in Section 4.2 broke the work into chunks, with chunk size being the number vertices divided by the number of threads.

For the Dijkstra algorithm, for instance, we could have this:

```
...
         #pragma omp for schedule(static,chunk)
         for (i = 1; i < nv; i++)  {
            if (notdone[i] && mind[i] < mymd)  {
               mymd = ohd[i];
               mymv = i;
            }
         }
...
         #pragma omp for schedule(static,chunk)
         for (i = 1; i < nv; i++)
            if (mind[mv] + ohd[mv*nv+i] < mind[i])
               mind[i] = mind[mv] + ohd[mv*nv+i];
...
```

But one can enhance performance by considering other chunk sizes (in which case a thread would be responsible for more than one chunk). On the one hand, large chunks are good, due to there being less overhead—every time a thread finishes a chunk, it must go through the critical section, which serializes our parallel program and thus slows things down. On the other hand, if chunk sizes are large, then toward the end of the work, some threads may be working on their last chunks while others have finished and are now idle, thus foregoing potential speed enhancement. So it would be nice to have large chunks at the beginning of the run, to reduce the overhead, but smaller chunks at the end. This can be done using the **guided** clause.

For the Dijkstra algorithm, for instance, we could have this:

```
...
         #pragma omp for schedule(guided)
         for (i = 1; i < nv; i++)  {
            if (notdone[i] && mind[i] < mymd)  {
               mymd = ohd[i];
               mymv = i;
            }
         }
...
         #pragma omp for schedule(guided)
         for (i = 1; i < nv; i++)
            if (mind[mv] + ohd[mv*nv+i] < mind[i])
               mind[i] = mind[mv] + ohd[mv*nv+i];
...
```

4.3.4 The OpenMP `reduction` Clause

The name of this OpenMP clause alludes to the term **reduction** in functional programming. Many parallel programming languages include such operations, to enable the programmer to more conveniently (and often more efficiently) have threads/processors cooperate in computing sums, products, etc. OpenMP does this via the **reduction** clause.

For example, consider

```
int z;
...
#pragma omp for reduction(+:z)
for (i = 0; i < n; i++)  z += x[i];
```

The pragma says that the threads will share the work as in our previous discussion of the **for** pragma. In addition, though, there will be independent copies of **z** maintained for each thread, each initialized to 0 before the loop begins. When the loop is entirely done, the values of **z** from the various threads will be summed, of course in an atomic manner.

Note that the **+** operator not only indicates that the values of **z** are to be summed, but also that their initial values are to be 0. If the operator were *****, say, then the product of the values would be computed, and their initial values would be 1.

Our use of the **reduction** clause here makes our programming much easier. Indeed, if had old serial code that we wanted to parallelize, we would have to make no change to it! OpenMP is taking care of both the work splitting across values of **i**, and the atomic operations. Moreover—note this carefully—it is efficient, because by maintaining separate copies of **z** until the loop is done, we are reducing the number of serializing atomic actions, and are avoiding time-costly cache coherency transactions and the like.

Without this construct, we would have to do

```
int z,myz=0;
...
#pragma omp for private(myz)
for (i = 0; i < n; i++)  myz += x[i];
#pragma omp critical
{ z += myz; }
```

Here are the eligible operators and the corresponding initial values:

In C/C++, you can use **reduction** with +, -, *, &, |, && and || (and the exclusive-or operator).

operator	initial value
+	0
-	0
*	1
&	bit string of 1s
|	bit string of 0s
ˆ	0
&&	1
||	0

The lack of other operations typically found in other parallel programming languages, such as min and max, is due to the lack of these operators in C/C++. The FORTRAN version of OpenMP does have min and max.[3]

4.4 The Task Directive

This is new to OpenMP 3.0. The basic idea is this: When a thread encounters a **task** directive, it arranges for some thread to execute the associated block. The first thread can continue. Note that the task might not execute right away; it may have to wait for some thread to become free after finishing another task.

Here's a Quicksort example:

```
// OpenMP example program:  quicksort; not necessarily efficient

void swap(int *yi, int *yj)
{  int tmp = *yi;
   *yi = *yj;
   *yj = tmp;
}

int *separate(int *x, int low, int high)
{  int i,pivot,last;
```

[3]Note, though, that plain min and max would not help in our Dijkstra example above, as we not only need to find the minimum value, but also need the vertex which attains that value.

```
   pivot = x[low];  // would be better to take, e.g., median of 1st 3 elts
   swap(x+low,x+high);
   last = low;
   for (i = low; i < high; i++) {
      if (x[i] <= pivot) {
         swap(x+last,x+i);
         last += 1;
      }
   }
   swap(x+last,x+high);
   return last;
}

// quicksort of the array z, elements zstart through zend; set the
// latter to 0 and m-1 in first call, where m is the length of z;
// firstcall is 1 or 0, according to whether this is the first of the
// recursive calls
void qs(int *z, int zstart, int zend, int firstcall)
{
   #pragma omp parallel
   {  int part;
      if (firstcall == 1) {
         #pragma omp single nowait
         qs(z,0,zend,0);
      } else {
         if (zstart < zend) {
            part = separate(z,zstart,zend);
            #pragma omp task
            qs(z,zstart,part-1,0);
            #pragma omp task
            qs(z,part+1,zend,0);
         }

      }
   }
}

main(int argc, char**argv)
{  int i,n,*w;
   n = atoi(argv[1]);
   w = malloc(n*sizeof(int));
   for (i = 0; i < n; i++) w[i] = rand();
   qs(w,0,n-1,1);
   if (n < 25)
      for (i = 0; i < n; i++) printf("%d\n",w[i]);
}
```

The code

```
if (firstcall == 1) {
   #pragma omp single nowait
   qs(z,0,zend,0);
```

gets things going. We want only one thread to execute the root of the recursion tree, hence the need for the **single** clause. After that, the code

```
part = separate(z,zstart,zend);
#pragma omp task
qs(z,zstart,part-1,0);
```

sets up a call to a subtree, with the **task** directive stating, "OMP system, please make sure that this subtree is handled by some thread."

This really simplifies the programming. Compare this to the Python **multiprocessing** version in Section 3.5, where the programmer needed to write code to handle the work queue.

There are various refinements, such as the barrier-like **taskwait** clause.

4.5 Other OpenMP Synchronization Issues

Earlier we saw the **critical** and **barrier** constructs. There is more to discuss, which we do here.

4.5.1 The OpenMP `atomic` Clause

The **critical** construct not only serializes your program, but also it adds a lot of overhead. If your critical section involves just a one-statement update to a shared variable, e.g.

```
x += y;
```

etc., then the OpenMP compiler can take advantage of an atomic hardware instruction, e.g. the LOCK prefix on Intel, to set up an extremely efficient critical section, e.g.

```
#pragma omp atomic
x += y;
```

Since it is a single statement rather than a block, there are no braces.

The eligible operators are:

```
++, --, +=, *=, <<=, &=, |=
```

4.5.2 Memory Consistency and the `flush` Pragma

Consider a shared-memory multiprocessor system with coherent caches, and a shared, i.e. global, variable **x**. If one thread writes to **x**, you might think that the cache coherency system will ensure that the new value is visible to other threads. But it is is not quite so simple as this.

For example, the compiler may store **x** in a register, and update **x** itself at certain points. In between such updates, since the memory location for **x** is not written to, the cache will be unaware of the new value, which thus will not be visible to other threads. If the processors have write buffers etc., the same problem occurs.

In other words, we must account for the fact that our program could be run on different kinds of hardware with different memory consistency models. Thus OpenMP must have its own memory consistency model, which is then translated by the compiler to mesh with the hardware.

OpenMP takes a **relaxed consistency** approach, meaning that it forces updates to memory ("flushes") at all synchronization points, i.e. at:

- **barrier**
- entry/exit to/from **critical**
- entry/exit to/from **ordered**
- entry/exit to/from **parallel**
- exit from **parallel for**
- exit from **parallel sections**
- exit from **single**

In between synchronization points, one can force an update to **x** via the **flush** pragma:

```
#pragma omp flush (x)
```

The flush operation is obviously architecture-dependent. OpenMP compilers will typically have the proper machine instructions available for some common architectures. For the rest, it can force a flush at the hardware level by doing lock/unlock operations, though this may be costly in terms of time.

4.6 Compiling, Running and Debugging OpenMP Code

4.6.1 Compiling

There are a number of open source compilers available for OpenMP, including:

- Omni: This is available at (`http://phase.hpcc.jp/Omni/`). To compile an OpenMP program in **x.c** and create an executable file **x**, run

```
omcc -g -o x x.c
```

- Ompi: You can download this at `http://www.cs.uoi.gr/~ompi/index.html`. Compile **x.c** by

  ```
  ompicc -g -o x x.c
  ```

- GCC, version 4.2 or later:[4] Compile **x.c** via

  ```
  gcc -fopenmp -g -o x x.c
  ```

4.6.2 Running

Just run the executable as usual.

The number of threads will be the number of processors, by default. To change that value, set the OMP_NUM_THREADS environment variable. For example, to get four threads in the C shell, type

```
setenv OMP_NUM_THREADS 4
```

4.6.3 Debugging

OpenMP's use of pragmas makes it difficult for the compilers to maintain your original source code line numbers, and your function and variable names. But with a little care, a symbolic debugger such as GDB can still be used. Here are some tips for the compilers mentioned above, using GDB as our example debugging tool:

- Omni: The function **main()** in your executable is actually in the OpenMP library, and your function **main()** is renamed **_ompc_main()**. So, when you enter GDB, first set a breakpoint at your own code:

  ```
  (gdb) b _ompc_main
  ```

  Then run your program to this breakpoint, and set whatever other breakpoints you want.

  You should find that your other variable and function names are unchanged.

- Ompi: During preprocessing of your file **x.c**, the compiler produces a file **x_ompi.c**, and the latter is what is actually compiled. Your function **main** is renamed to **_ompi_originalMain()**. Your other functions and variables are renamed. For example in our Dijkstra code, the function **dowork()** is renamed to **dowork_parallel_0**. And by the way, all indenting is lost! Keep these points in mind as you navigate through your code in GDB.

[4]You may find certain subversions of GCC 4.1 can be used too.

- GCC: GCC maintains line numbers and names well. In earlier versions, it had a problem in that it did not not retain names of local variables within blocks controlled by **omp parallel** at all. That problem is now fixed (e.g. in version 4.4 of the GCC suite).

4.7 Combining Work-Sharing Constructs

In our examples of the **for** pragma above, that pragma would come within a block headed by a **parallel** pragma. The latter specifies that a team of theads is to be created, with each one executing the given block, while the former specifies that the various iterations of the loop are to be distributed among the threads. As a shortcut, we can combine the two pragmas:

```
#pragma omp parallel for
```

This also works with the **sections** pragma.

4.8 Performance

As is usually the case with parallel programming, merely parallelizing a program won't necessarily make it faster, even on shared-memory hardware. Operations such as critical sections, barriers and so on serialize an otherwise-parallel program, sapping much of its speed. In addition, there are issues of cache coherency transactions, false sharing etc.

4.8.1 The Effect of Problem Size

To illustrate this, I ran our original Dijkstra example (Section 4.2 on various graph sizes, on a quad core machine. Here are the timings:

nv	nth	time
1000	1	0.005472
1000	2	0.011143
1000	4	0.029574

The more parallelism we had, the *slower* the program ran! The synchronization overhead was just too much to be compensated by the parallel computation.

However, parallelization did bring benefits on larger problems:

nv	nth	time
25000	1	2.861814
25000	2	1.710665
25000	4	1.453052

4.8.2 Some Fine Tuning

How could we make our Dijkstra code faster? One idea would be to eliminate the critical section. Recall that in each iteration, the threads compute their local minimum distance values **md** and **mv**, and then update the global values **md** and **mv**. Since the update must be atomic, this causes some serialization of the program. Instead, we could have the threads store their values **mymd** and **mymv** in a global array **mymins**, with each thread using a separate pair of locations within that array, and then at the end of the iteration we could have just one task scan through **mymins** and update **md** and **mv**.

Here is the resulting code:

```
// Dijkstra.c

// OpenMP example program:  Dijkstra shortest-path finder in a
// bidirectional graph; finds the shortest path from vertex 0 to all
// others

// **** in this version, instead of having a critical section in which
// each thread updates md and mv, the threads record their mymd and mymv
// values in a global array mymins, which one thread then later uses to
// update md and mv

// usage:  dijkstra nv print

// where nv is the size of the graph, and print is 1 if graph and min
// distances are to be printed out, 0 otherwise

#include <omp.h>

// global variables, shared by all threads by default

int nv,  // number of vertices
    *notdone, // vertices not checked yet
    nth,  // number of threads
    chunk,  // number of vertices handled by each thread
    md,  // current min over all threads
    mv,  // vertex which achieves that min
    largeint = -1;  // max possible unsigned int

int *mymins;  // (mymd,mymv) for each thread; see dowork()

unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
         // ohd[i*nv+j]
         *mind;  // min distances found so far

void init(int ac, char **av)
{  int i,j,tmp;
```

```
   nv = atoi(av[1]);
   ohd = malloc(nv*nv*sizeof(int));
   mind = malloc(nv*sizeof(int));
   notdone = malloc(nv*sizeof(int));
   // random graph
   for (i = 0; i < nv; i++)
      for (j = i; j < nv; j++)  {
         if (j == i) ohd[i*nv+i] = 0;
         else  {
            ohd[nv*i+j] = rand() % 20;
            ohd[nv*j+i] = ohd[nv*i+j];
         }
      }
   for (i = 1; i < nv; i++)  {
      notdone[i] = 1;
      mind[i] = ohd[i];
   }
}

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

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

void dowork()
{
   #pragma omp parallel
   {  int startv,endv,  // start, end vertices for my thread
          step,  // whole procedure goes nv steps
          me,
          mymv;  // vertex which attains the min value in my chunk
          unsigned mymd;  // min value found by this thread
      int i;
      me = omp_get_thread_num();
      #pragma omp single
      {  nth = omp_get_num_threads();
         if (nv % nth != 0) {
            printf("nv must be divisible by nth\n");
            exit(1);
         }
         chunk = nv/nth;
         mymins = malloc(2*nth*sizeof(int));
      }
      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
         findmymin(startv,endv,&mymd,&mymv);
         mymins[2*me] = mymd;
         mymins[2*me+1] = mymv;
         #pragma omp barrier
         // mark new vertex as done
         #pragma omp single
         {  md = largeint; mv = 0;
            for (i = 1; i < nth; i++)
               if (mymins[2*i] < md) {
                  md = mymins[2*i];
                  mv = mymins[2*i+1];
               }
            notdone[mv] = 0;
         }
         // now update my section of mind
         updatemind(startv,endv);
         #pragma omp barrier
      }
   }
}

int main(int argc, char **argv)
{  int i,j,print;
   double startime,endtime;
   init(argc,argv);
   startime = omp_get_wtime();
   // parallel
   dowork();
   // back to single thread
   endtime = omp_get_wtime();
   printf("elapsed time:  %f\n",endtime-startime);
   print = atoi(argv[2]);
   if (print)  {
      printf("graph weights:\n");
      for (i = 0; i < nv; i++)  {
         for (j = 0; j < nv; j++)
            printf("%u  ",ohd[nv*i+j]);
         printf("\n");
      }
      printf("minimum distances:\n");
      for (i = 1; i < nv; i++)
         printf("%u\n",mind[i]);
   }
}
```

Let’s take a look at the latter part of the code for one iteration;

```
         findmymin(startv,endv,&mymd,&mymv);
         mymins[2*me] = mymd;
         mymins[2*me+1] = mymv;
         #pragma omp barrier
```

```
        // mark new vertex as done
        #pragma omp single
        {  notdone[mv] = 0;
           for (i = 1; i < nth; i++)
              if (mymins[2*i] < md) {
                 md = mymins[2*i];
                 mv = mymins[2*i+1];
              }
        }
        // now update my section of mind
        updatemind(startv,endv);
        #pragma omp barrier
```

The call to **findmymin()** is as before; this thread finds the closest vertex to 0 among this thread's range of vertices. But instead of comparing the result to **md** and possibly updating it and **mv**, the thread simply stores its **mymd** and **mymv** in the global array **mymins**. After all threads have done this and then waited at the barrier, we have just one thread update **md** and **mv**.

Let's see how well this tack worked:

nv	nth	time
25000	1	2.546335
25000	2	1.449387
25000	4	1.411387

This brought us about a 15% speedup in the two-thread case, though less for four threads.

What else could we do? Here are a few ideas:

- False sharing could be a problem here. To address it, we could make **mymins** much longer, changing the places at which the threads write their data, leaving most of the array as padding.
- We could try the modification of our program in Section 4.3.1, in which we use the OpenMP **for** pragma, as well as the refinements stated there, such as **schedule**.
- We could try combining all of the ideas here.

4.8.3 OpenMP Internals

We may be able to write faster code if we know a bit about how OpenMP works inside.

You can get some idea of this from your compiler. For example, if you use the **-t** option with the Omni compiler, or **-k** with Ompi, you can inspect the result of the preprocessing of the OpenMP pragmas.

Here for instance is the code produced by Omni from the call to **findmymin()** in our Dijkstra program:

```
# 93 "Dijkstra.c"
findmymin(startv,endv,&(mymd),&(mymv));{
_ompc_enter_critical(&__ompc_lock_critical);
# 96 "Dijkstra.c"
if((mymd)<(((unsigned )(md)))){

# 97 "Dijkstra.c"
(md)=(((int )(mymd)));
# 97 "Dijkstra.c"
(mv)=(mymv);
}_ompc_exit_critical(&__ompc_lock_critical);
```

Fortunately Omni saves the line numbers from our original source file, but the pragmas have been replaced by calls to OpenMP library functions.

The document, *The GNU OpenMP Implementation*, `http://pl.postech.ac.kr/~gla/cs700-07f/ref/openMp/libgomp.pdf`, includes good outline of how the pragmas are translated.

4.9 Further Examples

There are additional CUDA examples in later sections of this book.

Chapter 5

Introduction to GPU Programming with CUDA

Even if you don't play video games, you can be grateful to the game players, as their numbers have given rise to a class of highly powerful parallel processing devices—**graphics processing units** (GPUs). Yes, you program right on the video card in your computer, even though your program may have nothing to do with graphics.

5.1 Overview

The video game market is so lucrative that the industry has developed ever-faster GPUs, in order to handle ever-faster and ever-more visually detailed video games. These actually are parallel processing hardware devices, so around 2003 some people began to wonder if one might use them for parallel processing of nongraphics applications.

Originally this was cumbersome. One needed to figure out clever ways of mapping one's application to some kind of graphics problem. Though some high-level interfaces were developed to automate this transformation, effective coding required some understanding of graphics principles.

But current-generation of GPUs separate out the graphics operations, and now consist of multiprocessor elements that run under the familiar shared-memory threads model. Thus they are easily programmable. Granted, effective coding still requires an intimate knowledge of the hardwre, but at least it's (more or less) familiar hardware, not requiring knowledge of graphics.

Moreover, unlike a multicore machine, with the ability to run just a few threads at one time, e.g. four threads on a quad core machine, GPUs can run *hundreds or thousands* of threads at once. There are various restrictions that come with this, but you can see that there is fantastic potential for speed here.

NVIDIA has developed the CUDA language as a vehicle for programming on their GPUs. It's basically just a slight extension of C, and has become very popular. More recently, the OpenCL language has been developed by Apple, AMD and others (including NVIDIA). It too is a slight extension of C, and it aims to provide a uniform interface that works with multicore machines in addition to GPUs.

Our discussion here focuses on CUDA and NVIDIA GPUs.

Some terminology:

- A CUDA program consists of code to be run on the **host**, i.e. the CPU, and code to run on the **device**, i.e. the GPU.
- A function that is called by the host to execute on the device is called a **kernel**.
- Threads in an application are grouped into **blocks**. The entirety of blocks is called the **grid** of that application.

5.2 Sample Program

Here's a sample program. And I've kept the sample simple: It just finds the sums of all the rows of a matrix.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// CUDA example:  finds row sums of an integer matrix m

// find1elt() finds the rowsum of one row of the nxn matrix m, storing the
// result in the corresponding position in the rowsum array rs; matrix
// stored as 1-dimensional, row-major order

__global__ void find1elt(int *m, int *rs, int n)
{
   int rownum = blockIdx.x;  // this thread will handle row # rownum
   int sum=0;
   for (int k = 0; k < n; k++)
      sum += m[rownum*n+k];
   rs[rownum] = sum;
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);  // number of matrix rows/cols
    int *hm, // host matrix
        *dm, // device matrix
        *hrs, // host rowsums
        *drs; // device rowsums
    int msize = n * n * sizeof(int);  // size of matrix in bytes
    // allocate space for host matrix
    hm = (int *) malloc(msize);
    // as a test, fill matrix with consecutive integers
    int t = 0,i,j;
    for (i = 0; i < n; i++) {
       for (j = 0; j < n; j++) {
          hm[i*n+j] = t++;
       }
    }
    // allocate space for device matrix
    cudaMalloc((void **)&dm,msize);
    // copy host matrix to device matrix
    cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
    // allocate rowsum arrays
    int rssize = n * sizeof(int);
    hrs = (int *) malloc(rssize);
    cudaMalloc((void **)&drs,rssize);
    // set up parameters for threads structure
    dim3 dimGrid(n,1);
    dim3 dimBlock(1,1,1);
    // invoke the kernel
    find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
    // wait for kernel to finish
    cudaThreadSynchronize();
    // copy row vector from device to host
    cudaMemcpy(hrs,drs,rssize,cudaMemcpyDeviceToHost);
    // check results
    if (n < 10) for(int i=0; i<n; i++) printf("%d\n",hrs[i]);
    // clean up
    free(hm);
    cudaFree(dm);
    free(hrs);
    cudaFree(drs);
}
```

This is mostly C, with a bit of CUDA added here and there. Here's how the program works:

- Our **main()** runs on the host.
- Kernel functions are identified by **__global__ void**, are called by the host, and serve as entries to the device.

  We have only one kernel invocation here, but could have many, say with the output of one serving as input to the next.
- Other functions that will run on the device, called by functions running on the device, must be identified by **__device__**, e.g.

  ```
  __device__ int sumvector(float *x, int n)
  ```
- When a kernel is called, each thread runs it. Each thread receives the same arguments.

- The kernel may call other functions to run on the device, using data stored in the device memory. Such functions may in turn make such calls.
- Each block and thread has an ID, stored in the programmer-accessible structs **blockIdx** and **threadIdx**. We'll discuss the details later, but for now, we'll just note that here the statement

  ```
  int rownum = blockIdx.x;
  ```

  picks up the block number, which our code in this example uses to determine which row to sum.
- One calls **cudaMalloc()** to dynamically allocate space on the device's global memory. Execution of the statement

  ```
  cudaMalloc((void **)&drs,rssize);
  ```

  allocates space *on the device*, pointed to by **drs**, a variable in the *host's* address space.

  The space allocated by a **cudaMalloc()** call on the device is global to all kernels, and resides in the global memory of the device (details on memory types later).

  One can also allocate device memory statically. For example, the statement

  ```
  __device int z[100];
  ```

  appearing outside any function definition would allocate space on device global memory, with scope global to all kernels. However, it is not accessible to the host.
- Data is transferred to and from the host and device memories via **cudaMemcpy()**.
- Kernels return **void** values, so values are returned via a kernel's arguments.

  Note carefully that a call to the kernel doesn't block; it returns immediately. For that reason, the code above has a host barrier call, to avoid copying the results back to the host from the device before they're ready:

  ```
  cudaThreadSynchronize();
  ```

  On the other hand, if our code were to have another kernel call, say on the next line after

  ```
  find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
  ```

  and if some of the second call's input arguments were the outputs of the first call, there would be an implied barrier betwwen the two calls; the second would not start execution before the first finished.

  Calls like **cudaMemcpy()** do block until the operation completes.

  There is also a thread barrier available for the threads themselves, at the block level. The call is

  ```
  __syncthreads();
  ```

This can only be invoked by threads within a block, not across blocks.

- I've written the program so that each thread will handle one row of the matrix. Since I've chosen to store the matrix in one-dimensional form, and since the matrix is of size n x n, the loop

  ```
  for (int k = 0; k < n; k++)
     sum += m[rownum*n+k];
  ```

  will indeed traverse the n elements of row number **rownum**, and compute their sum. That sum is then placed in the proper element of the output array:

  ```
  rs[rownum] = sum;
  ```

- After the kernel returns, the host must copy the result back from the device memory to the host memory.

5.3 Understanding the Hardware Structure

Scorecards, get your scorecards here! You can't tell the players without a scorecard—classic cry of vendors at baseball games

Know thy enemy—Sun Tzu, *The Art of War*

The enormous computational potential of GPUs cannot be unlocked without an intimate understanding of the hardware. This of course is a fundamental truism in the parallel processing world, but it is acutely important for GPU programming. This section presents an overview of the hardware.

5.3.1 Processing Units

A GPU consists of a large set of **streaming multiprocessors** (SMs); you might say it's a multi-multiprocessor. Each SM consists of a number of **streaming processors** (SPs). It is important to understand the motivation for this hierarchy: Two threads located in different SMs cannot synchronize with each other in the barrier sense. Though this sounds like a negative at first, it is actually a great advantage, as the independence of threads in separate SMs means that the hardware can run faster. So, if the CUDA application programmer can write his/her algorithm so as to have certain independent chunks, and those chunks can be assigned to different SMs (we'll see how, shortly), then that's a "win."

Note that at present, word size is 32 bits. Thus for instance floating-point operations in hardware are single-precision.

5.3.2 Thread Operation

GPU operation is highly threaded, and again, understanding of the details of thread operation is key to good performance.

5.3.2.1 SIMT Architecture

The threads running within an SM *can* synchronize with each other, but there is further hierarchy.

Threads are partitioned into groups called **blocks**. The hardware will assign an entire block to a single SM, though several blocks can run in the same SM. The hardware will then divide a block into **warps**, 32 threads to a warp. Knowing that the hardware works this way, the programmer controls the block size and the number of blocks, and in general writes the code to take advantage of how the hardware works.

A key point is that *all the threads in a warp run the code in lockstep*. During the machine instruction fetch cycle, the same instruction will be fetched for all of the threads in the warp. Then in the execution cycle, each thread will either execute that particular instruction or execute nothing. The execute-nothing case occurs in the case of branches; see below. This is the classical **single instruction, multiple data** (SIMD) pattern used in some early special-purpose computers such as the ILLIAC; here it is called **single instruction, multiple thread** (SIMT).

5.3.2.2 The Problem of Thread Divergence

The SIMT nature of thread execution has major implications for performance. Consider what happens with if/then/else code. If some threads in a warp take the "then" branch and others go in the "else" direction, they cannot operate in lockstep. That means that some threads must wait while others execute. This renders the code at that point serial rather than parallel, a situation called **thread divergence**. As one CUDA Web tutorial points out, this can be a "performance killer."

In turn, the implication for writing CUDA code is that if you have an "embarrassingly parallel" application—parallelizable into independent chunks with very little communication between them—or your application has components with that property, you should arrange to have the chunks run in different blocks.

5.3.2.3 "OS in Hardware"

Each SM runs the threads on a timesharing basis, just like an operating system (OS). This timesharing is implemented in the hardware, though, not in software as in the OS case. Just as a process in an OS is given a fixed-length timeslice, so that processes take turns running, in a GPU's hardware OS, warps take turns running, with fixed-length timeslices.

Another difference between the usual software OS and a GPU's hardware OS is the following. With an OS, if a process reaches an input/output operation, the OS suspends the process while I/O is pending, even if its turn is not up. The OS then runs some other process instead, so as to avoid wasting CPU cycles during the long period of time needed for the I/O. With an SM, the analogous situation is a long memory operation, to global memory; if a a warp of threads needs to access global memory (including local memory), the SM will schedule some other warp while the memory access is pending.

The hardware support for threads is extremely good. A context switch takes very little time, and thread startup is fast too. Moreover, as noted above, the long latency of global memory can be solved by having a lot of threads that the hardware can timeshare to hide that latency. For these reasons, CUDA programmers typically employ a large number of threads, each of which does only a small amount of work.

5.3.3 Memory Structure

Yet another key hierarchy is memory structure. Let's discuss the most important two types of memory first—shared and global. Here is a summary.

5.3.3.1 Shared and Global Memory

Here is a summary:

type	shared	global
scope	glbl. to block	glbl. to app.
size	small	large
loc.	on-chip	off-chip
speed	blinding	molasses
lifetime	kernel	application
host access?	no	yes
cached?	no	no

In prose form:

- Shared memory: All the threads in an SM share this memory, and use it to communicate, just as is the case with threads in CPUs. Access is very fast, as this memory is on-chip. It is declared inside the kernel, or in the kernel call.

  On the other hand, shared memory is small, 16K bytes, and the data stored in it are valid only for the life of the currently-executing kernel. Also, shared memory cannot be accessed directly by the host.

- Global memory: This is shared by all the threads in an entire application, and is persistent across kernel calls, throughout the life of the application, i.e. until the program running on the host exits. It is usually much larger than shared memory. It is accessible from the host. Pointers to global memory can (but do not have to) be declared outside the kernel, and in this case the bindings persist across kernel invocations as well.

On the other hand, it is off-chip and very slow, taking hundreds of clock cycles per access instead of just a few. This can be ameliorated, as will be discussed later.

The reader should pause here and reread the above comparison between shared and global memories. *The key implication is that shared memory is used essentially as a programmer-managed cache.*[1] Data will start out in global memory, but if a variable is to be accessed multiple times by the GPU code, it's better for the programmer to write code that copies it to shared memory, and then access the copy instead of the original. If the variable is changed (and is to be used further), the programmer must include code to copy it back to global memory.

Neither memory type is (hardware) cached.

Accesses to global and shared memory are done via half-warps, i.e. an attempt is made to do all memory accesses in a half-warp simultaneously. In that sense, only threads in a half-warp run simultaneously, but the full warp is *scheduled* simultaneously by the hardware OS.

The host can access global memory via **cudaMemcpy()**, as seen earlier. It cannot access shared memory. Here is a typical pattern:

```
__global__ void abckernel(int *abcglobalmem)
{
   __shared__ int abcsharedmem[100];
   // ... code to copy some of abcglobalmem to some of abcsharedmem
   // ... code for computation
   // ... code to copy some of abcsharedmem to some of abcglobalmem
}
```

Typically you would write the code so that each thread deals with its own portion of the shared data, e.g. its own portion of **abcsharedmem** and **abcglobalmem** above. However, all the threads in that block can read/write any element in **abcsharedmem**, while they are within **abckernel()**. While inside another function running on the device, say one called by **abckernel()**, then **abcsharedmem** is out of scope.

Shared memory consistency (recall Section 2.5) is sequential within a thread, but **relaxed** among threads in a block. A write by one thread is not guaranteed to be visible to the others until **__syncthreads()** is called. On the other hand, writes by a thread *will* be visible to that same thread in subsequent reads. Among the implications of this is that if each thread writes only to portions of shared memory that are not read by other threads in the block, then **__syncthreads()** need not be called.

It is also possible to allocate shared memory in the kernel call, along with the block and thread configuration. Here is an example:

[1]Global memory itself has no cache.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// CUDA example:  illustrates dynamically-allocated shared memory; does
// nothing useful, just copying an array from host to device global,
// then to device shared, doubling it there, then copying back to device
// global then host

__global__ void doubleit(int *dv, int n)
{  extern __shared__ int sv[];
   int me = threadIdx.x;
   // threads share in copying dv to sv
   sv[me] = 2 * dv[me];
   __syncthreads();  // probably not needed in this case
   dv[me] = sv[me];
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);  // number of matrix rows/cols
    int *hv, // host vector
        *dv; // device vector
    int vsize = n * sizeof(int);  // size of vector in bytes
    // allocate space for host vector
    hv = (int *) malloc(vsize);
    int t = 0,i;
    for (i = 0; i < n; i++)
       hv[i] = t++;
    // allocate space for device vector
    cudaMalloc((void **)&dv,vsize);
    // copy host matrix to device vector
    cudaMemcpy(dv,hv,vsize,cudaMemcpyHostToDevice);
    // set up parameters for threads structure
    dim3 dimGrid(1,1);
    dim3 dimBlock(n,1,1);
    // invoke the kernel
    doubleit<<<dimGrid,dimBlock,vsize>>>(dv,n);
    // wait for kernel to finish
    cudaThreadSynchronize();
    // copy row vector from device to host
    cudaMemcpy(hv,dv,vsize,cudaMemcpyDeviceToHost);
    // check results
    if (n < 10) for(int i=0; i<n; i++) printf("%d\n",hv[i]);
    // clean up
    free(hv);
    cudaFree(dv);
}
```

Here the variable **sv** is dynamically allocated. It's declared in the statement

```
extern __shared__ int sv[];
```

but actually allocated during the kernel invocation

```
doubleit<<<dimGrid,dimBlock,vsize>>>(dv,n);
```

in that third argument within the chevrons, **vsize**.

Note that one can only directly declare one array in this manner. However, one can make subarrays, e.g.

```
int *x = &sv[120];
```

would set up **x** as a subarray of **sv**.

5.3.3.2 Global-Memory Performance Issues

As noted, the **latency**—time to access a single word—for global memory is quite high, on the order of hundreds of clock cycles. However, the hardware attempts to ameliorate this problem in a couple of ways.

First, as mentioned earlier, if a warp has requested a global memory access that will take a long time, the harware will schedule another warp to run while the first is waiting. This is an example of a common parallel processing technique called **latency hiding**.

Second, the **bandwidth** to global memory—the number of words accessed per unit time—can be high, due to hardware actions called **coalescing**. This simply means that if the hardware sees that the threads in this half-warp (or at least the ones currently accessing global memory) are accessing consecutive words, the hardware can execute the memory requests in groups of up to 32 words at a time. This is true for both reads and writes.

The newer GPUs go even further, coalescing much more general access patterns, not just to consecutive words.

Global memory is organized into **partitions** of 256 bytes each. There will be six or eight partitions, depending on the GPU model.

5.3.3.3 Shared-Memory Performance Issues

Shared memory memory is divided into **banks**, in a low-order interleaved manner: Words with consecutive addresses are stored in consecutive banks, mod the number of banks, i.e. wrapping back to 0 when hitting the last bank. If for instance there are 8 banks, addresses 0, 8, 16,... will be in bank 0, addresses 1, 9, 17,... will be in bank 1 and so on.

The fact that all memory accesses in a half-warp are attempted simultaneously implies that the best access to shared memory arises when the accesses are to different banks. For this reason, if one needs to access an entire array in order, one should write one's code so that consecutive threads access consecutive array elements, so as to avoid bank conflicts, which inhibit speed.

An exception occurs if multiple threads wish to read from the same word in the same bank, in which case the word will be **broadcast** to all the requestors simultaneously.

5.3.3.4 Host/Device Memory Transfer Performance Issues

Copying data between host and device can be a major bottleneck. One way to ameliorate this is to use **cudaMallocHost()** instead of **malloc()** when allocating memory on the host. This sets up page-locked memory, said to make **cudaMemcpy()** twice as fast.

5.3.3.5 Other Types of Memory

There are also other types of memory. Again, let's start with a summary:

type	registers	local	constant
scope	single thread	single thread	glbl. to app.
loc.	on-chip	off-chip	off-chip
speed	blinding	molasses	molasses
lifetime	kernel	kernel	application
host access?	no	no	yes
cached?	no	no	yes

- Registers: Each SM has a set of registers. They are much more numerous than in a CPU. Access to them is very fast, said to be slightly faster than to shared memory.

  The compiler normally stores the local variables for a device function in registers, but there are exceptions. An array won't be placed in registers if the array is too large, or if the array has variable index values, such as

  ```
  int z[20],i;
  ...
  y = z[i];
  ```

  Since registers are not indexable, the compiler cannot allocate **z** to registers. If on the other hand, the only code accessing **z** has constant indices, e.g. **z[8]**, the compiler may put **z** in registers.

- Local memory: This is physically part of global memory, but is an area within that memory that is allocated by the compiler for a given thread. As such, it is slow, and accessible only by that thread. The compiler allocates this memory for local variables in a device function if the compiler cannot store them in registers. This is called **register spill**.

- Constant memory: As the name implies, it's read-only from the device (read/write by the host), for storing values that will not change. It is off-chip but has a cache on the chip. At present, the size is 64K.

  One designates this memory with **__constant__**, as a global variable in the source file. One sets its contents from the host via **cudaMemcpyToSymbol()**, For example:

  ```
  __constant__ int x;
  int y = 3;
  // host code
  cudaMemcpyToSymbol("x",&y,sizeof(int));
  ...
  // device code
  int z;
  z = x;
  ```

  Visible to all threads.

- Texture memory: This memory is closer to graphics applications. Read-only. It also is off-chip but has an on-chip cache.

Local variables, including fixed-length arrays, are allocated by the compiler to registers if possible, otherwise to local memory.

5.3.4 Threads Hierarchy

Like the hardware, threads in CUDA software follow a hierarchy:

- The entirety of threads for an application is called a **grid**.
- A grid consists of one or more **blocks** of threads.
- Each block has its own ID within the grid, consisting of an "x coordinate" and a "y coordinate."
- Likewise each thread has x, y and z coordinates within whichever block it belongs to.
- Just as an ordinary CPU thread needs to be able to sense its ID, e.g. by calling **omp_get_thread_num()** in OpenMP, CUDA threads need to do the same. A CUDA thread can access its block ID via the built-in variables **blockIdx.x** and **blockIdx.y**, and can access its thread ID within its block via **threadIdx.x**, **threadIdx.y** and **threadIdx.z**.
- The programmer specifies the grid size (the numbers of rows and columns of blocks within a grid) and the block size (numbers of rows, columns and layers of threads within a block). In the example above, this was done by the code

```
dim3 dimGrid(n,1);
dim3 dimBlock(1,1,1);
find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
```

Here the grid is specified to consist of n ($n \times 1$) blocks, and each block consists of just one ($1 \times 1 \times 1$) threads.

That last line is of course the call to the kernel. As you can see, CUDA extends C syntax to allow specifying the grid and block sizes. CUDA will store this information in structs **gridDim** and **blockDim**, accessible to the programmer, again with member variables for the various dimensions, e.g. **threadDim.x** for the size of the X dimension for the number of threads per block.

- As noted, all threads in a block run in the same SM, though more than one block might be on the same SM.
- The "coordinates" of a block within the grid, and of a thread within a block, are merely abstractions. *They do not correspond to any physical arrangment.*

The motivation for the two-dimensional block arrangment is to make coding conceptually simpler for the programmer if he/she is working an application that is two-dimensional in nature.

For example, in a matrix application one's parallel algorithm might be based on partitioning the matrix into rectangular submatrices (**tiles**), as we'll do in Section 8.2. In a small example there, the matrix

$$A = \left(\begin{array}{ccc} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{array} \right) \tag{5.1}$$

is partitioned as

$$A = \left(\begin{array}{cc} A_{00} & A_{01} \\ A_{10} & A_{11} \end{array} \right), \tag{5.2}$$

where

$$A_{00} = \left(\begin{array}{cc} 1 & 5 \\ 0 & 3 \end{array} \right), \tag{5.3}$$

$$A_{01} = \left(\begin{array}{c} 12 \\ 6 \end{array} \right), \tag{5.4}$$

$$A_{10} = \left(\begin{array}{cc} 4 & 8 \end{array} \right) \tag{5.5}$$

and

$$A_{11} = \left(\begin{array}{c} 2 \end{array} \right). \tag{5.6}$$

We might then have one block of threads handle A_{00}, another block handle A_{01} and so on. CUDA's two-dimensional ID system for blocks makes life easier for programmers in such situations.

5.3.5 What's NOT There

We're not in Kansas anymore, Toto—character Dorothy Gale in *The Wizard of Oz*

It looks like C, it feels like C, and for the most part, it *is* C. But in many ways, it's quite different from what you're used to:

- You don't have access to the C library, e.g. **printf()** (the library consists of host machine language, after all). There are special versions of math functions, however, e.g. **__sin()**.
- No recursion.
- No stack. Functions are essentially inlined, rather than their calls being handled by pushes onto a stack.
- No pointers to functions.

5.4 Synchronization

As noted earlier, a barrier for the threads in the same block is available by calling **__syncthreads()**. Note carefully that if one thread writes a variable to shared memory and another then reads that variable, one must call this function in order to get the latest value. Remember, threads across blocks cannot sync with each other.

Several **atomic** operations—read/modify/write actions that a thread can execute without **pre-emption**, i.e. without interruption—are available on both global and shared memory. For example, **atomicAdd()** performs a fetch-and-add operation, as described in Section 2.6 of this book. The call is

```
atomicAdd(address of integer variable,inc);
```

where **address of integer variable** is the address of the (device) variable to add to, and **inc** is the amount to be added.

There are also **atomicExch()** (exchange the two operands), **atomicCAS()** (if the first operand equals the second, replace the first by the third), **atomicMin()**, **atomicMax()**, **atomicAnd()**, **atomicOr()**, and so on.

Use **-arch=sm_11** when compiling.

5.5 Hardware Requirements, Installation, Compilation, Debugging

You do need what is currently (March 2010) a high-end NVIDIA video card. There is a list at `http://www.nvidia.com/object/cuda_gpus.html`. If you have a Linux system, run **lspci** to determine what kind you have.

Download the CUDA toolkit from NVIDIA. Just plug "CUDA download" into a Web search engine to find the site. Install as directed.

You'll need to set your search and library paths to include the CUDA **bin** and **lib** directories.

To compile **x.cu** (and yes, use the **.cu** suffix), type

```
$ nvcc -g -G x.cu -I/your_CUDA_include_path
```

The **-g -G** options are for setting up debugging, the first for host code, the second for device code. Run the code as you normally would.

You'll need to set your library path properly. For example, on Linux machines, set the environment variable **LD_LIBRARY_PATH** to include the CUDA library.

To determine the limits, e.g. maximum number of threads, for your device, use code like this:

```
cudaDeviceProp Props;
cudaGetDeviceProperties(Props,0);
```

The 0 is for device 0, assuming you only have one device. The return value of **cudaGetDeviceProperties()** is a complex C struct whose components are listed at `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/online/group__CUDART__DEVICE_g5aa4f47938af8276f08074` `html`. But I recommend printing it from within GDB to see the values. One of the fields gives clock speed, which is typically slower than that of the host.

Debug using GDB as usual. You must compile your program in emulation mode, using the **-deviceemu** command-line option. This usually should be good enough, but CUDA also includes a special version of GDB, CUDA-GDB (invoked as **cuda-gdb**) for real-time debugging. However, it runs only on Unix-family platforms, and even then, only if X11 is not running! So, GDB is the likely option of choice.

5.6 Improving the Sample Program

The issues involving coalescing in Section 5.3.3.2 would suggest that our rowsum code might run faster with column sums. As two threads in the same half-warp march down adjoining columns in lockstep, they will always be accessing adjoining words in memory.

So, I modified the program accordingly (not shown), and compiled the two versions, as **rs** and **cs**, the row- and column-sum versions of the code, respectively.

This did produce a small improvement (confirmed in subsequent runs, needed in any timing experiment):

```
pc5:~/CUDA% time rs 20000
2.585u 1.753s 0:04.54 95.3%     0+0k 7104+0io 54pf+0w
pc5:~/CUDA% time cs 20000
2.518u 1.814s 0:04.40 98.1%     0+0k 536+0io 5pf+0w
```

But let's compare it to a version running only on the CPU,

```
#include <stdio.h>
#include <stdlib.h>

// non-CUDA example:  finds col sums of an integer matrix m

// find1elt() finds the colsum of one col of the nxn matrix m, storing the
// result in the corresponding position in the colsum array cs; matrix
// stored as 1-dimensional, row-major order

void find1elt(int *m, int *cs, int n)
{
   int sum=0;
   int topofcol;
   int col,k;
   for (col = 0; col < n; col++) {
      topofcol = col;
      sum = 0;
      for (k = 0; k < n; k++)
         sum += m[topofcol+k*n];
      cs[col] = sum;
   }
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);  // number of matrix cols/cols
    int *hm, // host matrix
        *hcs; // host colsums
    int msize = n * n * sizeof(int);  // size of matrix in bytes
    // allocate space for host matrix
    hm = (int *) malloc(msize);
    // as a test, fill matrix with consecutive integers
    int t = 0,i,j;
    for (i = 0; i < n; i++) {
       for (j = 0; j < n; j++) {
          hm[i*n+j] = t++;
       }
    }
    int cssize = n * sizeof(int);
    hcs = (int *) malloc(cssize);
    find1elt(hm,hcs,n);
    if (n < 10) for(i=0; i<n; i++) printf("%d\n",hcs[i]);
    // clean up
    free(hm);
    free(hcs);
}
```

How fast does this non-CUDA version run?

```
pc5:~/CUDA% time csc 20000
61.110u 1.719s 1:02.86 99.9%    0+0k 0+0io 0pf+0w
```

Very impressive! No wonder people talk of CUDA in terms like "a supercomputer on our desktop." And remember, this includes the time to copy the matrix from the host to the device (and to copy the output array back). And we didn't even try to optimize thread configuration, memory coalescing and bank usage, making good use of memory hierarchy, etc.

On the other hand, remember that this is an "embarrassingly parallel" application, and in many applications we may have to settle for a much more modest increase, and work harder to get it.

5.7 More Examples

5.7.1 Finding the Mean Number of Mutual Outlinks

Consider a network graph of some kind, such as Web links. For any two vertices, say any two Web sites, we might be interested in mutual outlinks, i.e. outbound links that are common to two Web sites. The CUDA code below finds the mean number of mutual outlinks, among all pairs of sites in a set of Web sites.

```
#include <cuda.h>
#include <stdio.h>

// CUDA example:  finds mean number of mutual outlinks, among all pairs
// of Web sites in our set

// for a given thread number tn, returns pointer to pair, the (i,j) to be
// processed by that thread; for nxn matrix
__device__ void findpair(int tn, int n, int *pair)
{  int sum=0,oldsum=0,i;
   for(i=0; ;i++) {
      sum += n - i - 1;
      if (tn <= sum-1) {
         pair[0] = i;
         pair[1] = tn - oldsum + i + 1;
         return;
      }
      oldsum = sum;
   }
}

// proc1pair() processes one pair of Web sites, i.e. one pair of rows in
// the nxn adjacency matrix m; the number of mutual outlinks is added to
// tot
__global__ void proc1pair(int *m, int *tot, int n)
{
   // find (i,j) pair to assess for mutuality
   int pair[2];
   findpair(threadIdx.x,n,pair);
   int sum=0;
   int startrowa = pair[0] * n,
       startrowb = pair[1] * n;
   for (int k = 0; k < n; k++)
      sum += m[startrowa + k] * m[startrowb + k];
   atomicAdd(tot,sum);
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]);  // number of matrix rows/cols
    int *hm, // host matrix
        *dm, // device matrix
        htot, // host grand total
        *dtot; // device grand total
    int msize = n * n * sizeof(int);  // size of matrix in bytes
    // allocate space for host matrix
    hm = (int *) malloc(msize);
    // as a test, fill matrix with random 1s and 0s
    int i,j;
    for (i = 0; i < n; i++) {
       hm[n*i+i] = 0;
       for (j = 0; j < n; j++) {
          if (j != i) hm[i*n+j] = rand() % 2;
       }
    }
    // allocate space for device matrix
    cudaMalloc((void **)&dm,msize);
    // copy host matrix to device matrix
    cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
    htot = 0;
    // set up device total and initialize it
    cudaMalloc((void **)&dtot,sizeof(int));
    cudaMemcpy(dtot,&htot,sizeof(int),cudaMemcpyHostToDevice);
    // set up parameters for threads structure
    dim3 dimGrid(1,1);
    int npairs = n*(n-1)/2;
    dim3 dimBlock(npairs,1,1);
    // invoke the kernel
    proc1pair<<<dimGrid,dimBlock>>>(dm,dtot,n);
    // wait for kernel to finish
    cudaThreadSynchronize();
    // copy total from device to host
    cudaMemcpy(&htot,dtot,sizeof(int),cudaMemcpyDeviceToHost);
    // check results
    if (n <= 15) {
       for (i = 0; i < n; i++) {
          for (j = 0; j < n; j++)
             printf("%d ",hm[n*i+j]);
          printf("\n");
       }
    }
    printf("mean = %f\n",htot/float(npairs));
    // clean up
    free(hm);
    cudaFree(dm);
    cudaFree(dtot);
}
```

The main programming issue here is finding a way to partition the various pairs (i,j) to the different threads. The function **findpair()** here does that.

Note the use of **atomicAdd()**.

5.7.2 Finding Prime Numbers

The code below finds all the prime numbers from 2 to **n**.

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>

// CUDA example:  illustration of shared memory allocation at run time;
// finds primes using classical Sieve of Erathosthenes:  make list of
// numbers 2 to n, then cross out all multiples of 2 (but not 2 itself),
// then all multiples of 3, etc.; whatever is left over is prime; in our
// array, 1 will mean "not crossed out" and 0 will mean "crossed out"

// IMPORTANT NOTE: uses shared memory, in a single block, without
// rotating parts of array in and out of shared memory; thus limited to
// n <= 4000 if have 16K shared memory

// initialize sprimes, 1s for the odds, 0s for the evens; see sieve()
// for the nature of the arguments
__device__ void initsp(int *sprimes, int n, int nth, int me)
{
   int chunk,startsetsp,endsetsp,val,i;
   sprimes[2] = 1;
   // determine sprimes chunk for this thread to init
   chunk = (n-1) / nth;
   startsetsp = 2 + me*chunk;
   if (me < nth-1) endsetsp = startsetsp + chunk - 1;
   else endsetsp = n;
   // now do the init
   val = startsetsp % 2;
   for (i = startsetsp; i <= endsetsp; i++) {
      sprimes[i] = val;
      val = 1 - val;
   }
   // make sure sprimes up to date for all
   __syncthreads();
}

// copy sprimes back to device global memory; see sieve() for the nature
// of the arguments
__device__ void cpytoglb(int *dprimes, int *sprimes, int n, int nth, int me)
{
   int startcpy,endcpy,chunk,i;
   chunk = (n-1) / nth;
   startcpy = 2 + me*chunk;
   if (me < nth-1) endcpy = startcpy + chunk - 1;
   else endcpy = n;
   for (i = startcpy; i <= endcpy; i++) dprimes[i] = sprimes[i];
   __syncthreads();
}

// finds primes from 2 to n, storing the information in dprimes, with
// dprimes[i] being 1 if i is prime, 0 if composite; nth is the number
// of threads (threadDim somehow not recognized)
__global__ void sieve(int *dprimes, int n, int nth)
{
   extern __shared__ int sprimes[];
   int me = threadIdx.x;
   int nth1 = nth - 1;
   // initialize sprimes array, 1s for odds, 0 for evens
   initsp(sprimes,n,nth,me);
   // "cross out" multiples of various numbers m, with each thread doing
   // a chunk of m's; always check first to determine whether m has
   // already been found to be composite; finish when m*m > n
   int maxmult,m,startmult,endmult,chunk,i;
   for (m = 3; m*m <= n; m++) {
      if (sprimes[m] != 0) {
         // find largest multiple of m that is <= n
         maxmult = n / m;
         // now partition 2,3,...,maxmult among the threads
         chunk = (maxmult - 1) / nth;
         startmult = 2 + me*chunk;
         if (me < nth1) endmult = startmult + chunk - 1;
         else endmult = maxmult;
      }
      // OK, cross out my chunk
      for (i = startmult; i <= endmult; i++) sprimes[i*m] = 0;
   }
   __syncthreads();
   // copy back to device global memory for return to host
   cpytoglb(dprimes,sprimes,n,nth,me);
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]),  // will find primes among 1,...,n
```

```
      nth = atoi(argv[2]);  // number of threads
   int *hprimes,  // host primes list
      *dprimes;  // device primes list
   int psize = (n+1) * sizeof(int);  // size of primes lists in bytes
   // allocate space for host list
   hprimes = (int *) malloc(psize);
   // allocate space for device list
   cudaMalloc((void **)&dprimes,psize);
   dim3 dimGrid(1,1);
   dim3 dimBlock(nth,1,1);
   // invoke the kernel, including a request to allocate shared memory
   sieve<<<dimGrid,dimBlock,psize>>>(dprimes,n,nth);
   // check whether we asked for too much shared memory
   cudaError_t err = cudaGetLastError();
   if(err != cudaSuccess) printf("%s\n",cudaGetErrorString(err));
   // wait for kernel to finish
   cudaThreadSynchronize();
   // copy list from device to host
   cudaMemcpy(hprimes,dprimes,psize,cudaMemcpyDeviceToHost);
   // check results
   if (n <= 1000) for(int i=2; i<=n; i++)
      if (hprimes[i] == 1) printf("%d\n",i);
   // clean up
   free(hprimes);
   cudaFree(dprimes);
}
```

This code has been designed with some thought as to memory speed and thread divergence. Ideally, we would like to use device shared memory if possible, and to exploit the lockstep, SIMD nature of the hardware.

The code uses the classical Sieve of Erathosthenes, "crossing out" multiples of 2, 3, 5, 7 and so on to get rid of all the composite numbers. However, the code here differs from that in Section 1.3.1.2, even though both programs use the Sieve of Erathosthenes.

Say we have just two threads, A and B. In the earlier version, thread A might cross out all multiples of 19 while B handles multiples of 23. In this new version, thread A deals with only some multiples of 19 and B handles the others for 19. Then they both handle their own portions of multiples of 23, and so on. The thinking here is that the second version will be more amenable to lockstep execution, thus causing less thread divergence.

Thus in this new version, each thread handles a chunk of multiples of the given prime. Note the contrast of this with many CUDA examples, in which each thread does only a small amount of work, such as computing a single element in the product of two matrices.

In order to enhance memory performance, this code uses device shared memory. All the "crossing out" is done in the shared memory array **sprimes**, and then when we are all done, that is copied to the device global memory array **dprimes**, which is in turn copies to host memory. By the way, note that the amount of shared memory here is determined dynamically.

However, device shared memory consists only of 16K bytes, which would limit us here to values of **n** up to about 4000. Extending the program to work for larger values of **n** would require some careful planning if we still wish to use shared memory.

5.8 CUBLAS

CUDA includes some parallel linear algebra routines callable from straight C code. In other words, you can get the benefit of GPU in linear algebra contexts without using CUDA. Note, though, that in fact you *are* using CUDA, behind the scenes.

And indeed, you can mix CUDA and CUBLAS code. Your program might have multiple kernel invocations, some CUDA and others CUBLAS, with each using data in device global memory that was written by earlier kernels. Again, remember, the contents of device global memory (including the bindings of variable names) are persistent across kernel calls in the same application.

Below is an example **RowSumsCB.c**, the matrix row sums example again, this time using CUBLAS. We can find the vector of row sums of the matrix A by post-multiplying A by a column vector of all 1s.

I compiled the code by typing

```
gcc -g -I/usr/local/cuda/include -L/usr/local/cuda/lib RowSumsCB.c -lcublas -lcudart
```

You should modify for your own CUDA locations accordingly. Users who merely wish to use CUBLAS will find the above more convenient, but **nvcc** can be used too:

```
nvcc -g -G -I/usr/local/cuda/include RowSumsCB.c -lcublas
```

Here is the code:

```
#include <stdio.h>
#include <cublas.h>  // required include

int main(int argc, char **argv)
{
   int n = atoi(argv[1]);  // number of matrix rows/cols
   float *hm, // host matrix
         *hrs, // host rowsums vector
         *ones,  // 1s vector for multiply
         *dm, // device matrix
         *drs; // device rowsums vector
   // allocate space on host
   hm = (float *) malloc(n*n*sizeof(float));
   hrs = (float *) malloc(n*sizeof(float));
   ones = (float *) malloc(n*sizeof(float));
   // as a test, fill hm with consecutive integers, but in column-major
   // order for CUBLAS; also put 1s in ones
   int i,j;
   float t = 0.0;
   for (i = 0; i < n; i++) {
      ones[i] = 1.0;
      for (j = 0; j < n; j++)
         hm[j*n+i] = t++;
   }
   cublasInit();  // required init
   // set up space on the device
   cublasAlloc(n*n,sizeof(float),(void**)&dm);
   cublasAlloc(n,sizeof(float),(void**)&drs);
   // copy data from host to device
   cublasSetMatrix(n,n,sizeof(float),hm,n,dm,n);
   cublasSetVector(n,sizeof(float),ones,1,drs,1);
   // matrix times vector
   cublasSgemv('n',n,n,1.0,dm,n,drs,1,0.0,drs,1);
   // copy result back to host
   cublasGetVector(n,sizeof(float),drs,1,hrs,1);
   // check results
   if (n < 20) for (i = 0; i < n; i++) printf("%f\n",hrs[i]);
   // clean up on device (should call free() on host too)
   cublasFree(dm);
   cublasFree(drs);
   cublasShutdown();
}
```

As noted in the comments, CUBLAS assumes FORTRAN-style, i.e. column-major orrder, for matrices.

Now that you know the basic format of CUDA calls, the CUBLAS versions will look similar. In the call

```
cublasAlloc(n*n,sizeof(float),(void**)&dm);
```

for instance, we are allocating space on the device for an n x n matrix of **float**s, on the device.

The call

```
cublasSetMatrix(n,n,sizeof(float),hm,n,dm,n);
```

is slightly more complicated. Here we are saying that we are copying **hm**, an n x n matrix of **float**s on the host, to **dm** on the host. The **n** arguments in the last and third-to-last positions again say that the two matrices each have **n** rows. This seems redundant, but this is sometimes needed in cases of matrix tiling.

The 1s in the call

```
cublasSetVector(n,sizeof(float),ones,1,drs,1);
```

are needed for similar reasons. We are saying that in our source vector **ones**, for example, the elements of interest are spaced 1 elements apart, i.e. they are contiguous. But if we wanted our vector to be some row in a matrix with, say, 500 rows, the elements of interesting would be spaced 500 elements apart, again keeping in mind that column-major order is assumed.

The actual matrix multiplication is done here:

```
cublasSgemv('n',n,n,1.0,dm,n,drs,1,0.0,drs,1);
```

The “mv” in “cublasSgemv” stands for “matrix times vector.”

Further information is available in the CUBLAS manual.

5.9 Error Checking

Every CUDA call (except for kernel invocations) returns an error code of type **cudaError_t**. One can view the nature of the error by calling **cudaGetErrorString()** and printing its output.

For kernel invocations, one can call **cudaGetLastError()**, which does what its name implies. A call would typically have the form

```
cudaError_t err = cudaGetLastError();
if(err != cudaSuccess) printf("%s\n",cudaGetErrorString(err));
```

You may also wish to **cutilSafeCall()**, which is used by wrapping your regular CUDA call. It automatically prints out error messages as above.

Each CUBLAS call returns a potential error code, of type **cublasStatus**, not checked here.

5.10 Further Examples

There are additional CUDA examples in later sections of this book. These include:

- Matrix-multiply code, optimized for use of shared memory, in Section 8.3.2.2.

Chapter 6

Message Passing Systems

Message passing systems are probably the most common platforms for parallel processing today.

6.1 Overview

Traditionally, shared-memory hardware has been extremely expensive, with a typical system costing hundreds of thousands of dollars. Accordingly, the main users were for very large corporations or government agencies, 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.

But the huge expense of shared-memory machines led to a quest for high-performance message-passing alternatives, first in hypercubes and then in networks of workstations (NOWs).

The situation changed radically around 2005, when "shared-memory hardware for the masses" became available in dual-core commodity PCs. Chips of higher core multiplicity are commercially available, with a decline of price being inevitable. Ordinary users will soon be able to afford shared-memory machines featuring dozens of processors.

Yet the message-passing paradigm continues to thrive. Many people believe it is more amenable to writing really fast code, and the the advent of **cloud computing** has given message-passing a big boost. In addition, many of the world's very fastest systems (see `www.top500.org` for the latest list) are in fact of the message-passing type.

In this chapter, we take a closer look at this approach to parallel processing.

6.2 A Historical Example: Hypercubes

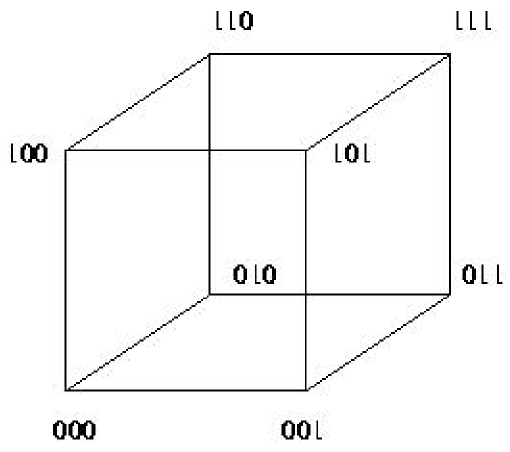
A popular class of parallel machines in the 1980s and early 90s was that of **hypercubes**. Intel sold them, for example, as did a subsidiary of Oracle, nCube. A hypercube would consist of some number of ordinary Intel processors, with each processor having some memory and serial I/O hardware for connection 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, both for historical reasons (in the computer field, old techniques are often recycled decades later), and because 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.

6.2.0.0.1 Definitions A **hypercube** of dimension d consists of $D = 2^d$ **processing elements** (PEs), i.e. processor-memory pairs, with fast serial I/O connections between neighboring PEs. 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 ith neighbor has number $(c_{d-1}, ..., 1 - c_{i-1}, ..., c_0)$.[1]

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.



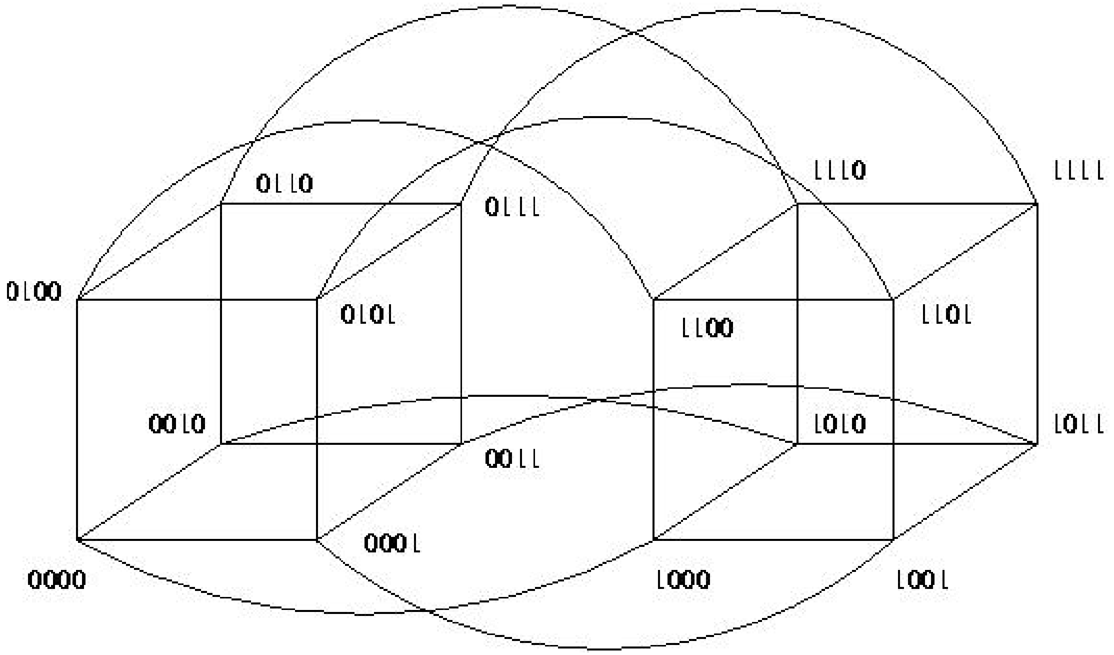
It is sometimes helpful to build up a cube from the lower-dimensional cases. To build a (d+1)-dimensional cube from two d-dimensional cubes, just follow this recipe:

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

(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.[2] 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

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

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.

6.3 Networks of Workstations (NOWs)

The idea here is simple: Take a bunch of commodity PCs 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 message-passing software environments such as MPI, we can form very powerful parallel systems.

The networking does result in a significant loss of performance, but the price/performance ratio in NOW can be much superior in many applications to that of shared-memory or hypercube hardware of comparable number of CPUs.

6.3.1 The Network Is Literally the Weakest Link

Still, 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 designers of the Internet, e.g. e-mail and file transfer, but they are slow in the NOW context.

A popular network for a NOW today is Infiniband (IB) (`www.infinibandta.org`). It features low latency, about 1.0-3.0 microseconds, high bandwidth, about 1.0-2.0 gigaBytes per second), and uses a low amount of the CPU's cycles, around 5-10%.

The basic building block of IB is a switch, with many inputs and outputs, similar in concept to Ω-net. You can build arbitrarily large and complex topologies from these switches.

A central point is that IB, as with other high-performance networks designed for NOWs, uses RDMA (Remote Direct Memory Access) read/write, which eliminates the extra copying of data between the application program's address space to that of the operating system.

IB has high performance and scalable[3] implementations of distributed locks, semaphores, collective communication operations. An atomic operation takes about 3-5 microseconds.

[3]The term *scalable* arises frequently in conversations on parallel processing. It means that this particular method of dealing with some aspect of parallel processing continues to work well as the system size increases. We say that the method *scales*.

IB implements true **multicast**, i.e. the simultaneous sending of messages to many nodes. Note carefully that even though MPI has its **MPI_Bcast()** function, it will send things out one at a time unless your network hardware is capable of multicast, and the MPI implementation you use is configured specifically for that hardware.

For information on network protocols, e.g. for example `www.rdmaconsortium.org`. A research paper evaluating a tuned implementation of MPI on IB is available at `nowlab.cse.ohio-state.edu/publications/journal-papers/2004/liuj-ijpp04.pdf`.

6.3.2 Other Issues

Increasingly today, the workstations themselves are multiprocessor machines, so a NOW really is a hybrid arrangement. They can be programmed either purely in a message-passing manner—e.g. running eight MPI processes on four dual-core machines—or in a mixed way, with a shared-memory approach being used within a workstation but message-passing used between them.

NOWs have become so popular that there are now "recipes" on how to build them for the specific purpose of parallel processing. The term **Beowulf** come to mean a NOW, usually with a fast network connecting them, used for parallel processing. The term *NOW* itself is no longer in use, replaced by *cluster*. Software packages such as ROCKS (`http://www.rocksclusters.org/wordpress/`) have been developed to make it easy to set up and administer such systems.

6.4 Systems Using Nonexplicit Message-Passing

Writing message-passing code is a lot of work, as the programmer must explicitly arrange for transfer of data. Contrast that, for instance, to shared-memory machines, in which cache coherency transactions will cause data transfers, but which are not arranged by the programmer and not even seen by him/her.

In order to make coding on message-passing machines easier, higher-level systems have been devised. These basically operate in the **scatter/gather** paradigm, in which a "manager" node sends out chunks of work to the other nodes, serving as "workers," and then collects and assembles the results sent back the workers.

One example of this is R's **snow** package, which will be discussed in Section 7.5.2.2. But the most common approach today—and the one attracting the most attention—is MapReduce, to be discussed below.

6.4.1 MapReduce

MapReduce was developed as part of a recently-popularized computational approach known as **cloud computing**. The idea is that a large corporation that has many computers could sell time on them, thus making profitable use of excess capacity. The typical customer would have occasional need for large-scale computing—and often large-scale data storage. The customer would submit a program to the cloud computing vendor, who would run it in parallel on the vendor's many machines (unseen, thus forming the "cloud"), then return the output to the customer.

Google, Yahoo! and Amazon, among others, have recently gotten into the cloud computing business. The open-source application of choice for this is Hadoop.

The key issue, of course, is the parallelizability of the inherently serial code. But all the user need do is provide code to break the data into chunks, code to work on a chunk, and code to collect the outputs from the chunks back into the overall output of the program.

For this to work, the program's data usage pattern must have a simple, regular structure, as in these examples:

Example 1: Suppose we wish to list all the words used in a file, together with the counts of the numbers of instances of the words. If we have 100000 lines in the file and 10 processors, we could divide the file into chunks of 10000 lines each, have each processor run code to do the word counts in its chunk, and then combine the results.

Example 2: Suppose we wish to multiply an nx1 vector X by an nxn matrix A. Say n = 100000, and again we have 10 processors. We could divide A into chunks of 10000 rows each, have each processor multiply X by its chunk, and then combine the outputs.

To illustrate this, here is a pseudocode summary of a word-count program written in Python by Michael Noll; see `http://www.michael-noll.com/wiki/Writing_An_Hadoop_MapReduce_Program_In_Python`. Actually Hadoop is really written for Java applications. However, Hadoop can work with programs in any language under Hadoop's Streaming option, by reading from STDIN and writing to STDOUT. This does cause some slowdown in numeric programs, for the conversion of strings to numbers and vice versa.[4]

mapper.py:

```
for each line in STDIN
   break line into words, placed in wordarray
   for each word in wordarray
      print word, '1' to STDOUT  # we have found 1 instance of the word
```

reducer.py:

```
# dictionary will consist of (word,count) pairs
dictionary = empty
for each line in STDIN
   split line into word, thiscount
```

[4]In the case of Python, we could also run Jython, a Python interpreter that produces Java byte code. Hadoop also offers communication via Unix pipes.

```
   if word not in dictionary:
      add (word,thiscount) to dictionary
   else
      change (word,count) entry to (word,count+thiscount)
print dictionary to STDOUT
```

Note that these two user programs have nothing in them at all regarding parallelism. Instead, the process works as follows:

- the user provides Hadoop the original data file, by copying the file to Hadoop's own file system, the Hadoop Distributed File System (HDFS)
- the user provides Hadoop with the mapper and reducer programs; Hadoop runs several instances of each
- Hadoop forms chunks by forming groups of lines in the file
- Hadoop has each instance of the mapper program work on a chunk:

  ```
  mapper.py < chunk > outputchunk
  # output is replicated and sent to the various instances of reducer
  ```

- Hadoop runs

  ```
  reducer.py < outputchunk > myfinalchunk
  # in this way final output is distributed to the nodes in HDFS
  ```

In the matrix-multiply model, the mapper program would produce chunks of X, together with the corresponding row numbers. Then the reducer program would sort the rows by row number, and place the result in X.

Note too that by having the file in HDFS, we minimize communications costs in shipping the data. "Moving computation is cheaper than moving data."

Hadoop also incorporates rather sophisticated fault tolerance mechanisms. If a node goes down, the show goes on.

Note again that this works well only on problems of a certain structure. Also, some say that the idea has been overpromoted; see for instance "MapReduce: A Major Step Backwards," *The Database Column*, by Professor David DeWitt, `http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html`

Chapter 7

Introduction to MPI

MPI is the *de facto* standard for message-passing software.

7.1 Overview

7.1.1 History

Though (small) shared-memory machines have come down radically in price, to the point at which a dual-core PC is affordable in the home, historically shared-memory machines were available only to the "very rich"—large banks, national research labs and so on.

The first "affordable" message-machine type was the Hypercube, developed by a physics professor at Cal Tech. It consisted of a number of **processing elements** (PEs) connected by fast serial I/O cards. This was in the range of university departmental research labs. It was later commercialized by Intel and NCube.

Later, the notion of **networks of workstations** (NOWs) became popular. Here the PEs were entirely independent PCs, connected via a standard network. This was refined a bit, by the use of more suitable network hardware and protocols, with the new term being **clusters**.

All of this necessitated the development of standardized software tools based on a message-passing paradigm. The first popular such tool was Parallel Virtual Machine (PVM). It still has its adherents today, but has largely been supplanted by the Message Passing Interface (MPI).

MPI itself later became MPI 2. Our document here is intended mainly for the original.

7.1.2 Structure and Execution

MPI is merely a set of Application Programmer Interfaces (APIs), called from user programs written in C, C++ and other languages. It has many implementations, with some being open source and generic, while others are proprietary and fine-tuned for specific commercial hardware.

Suppose we have written an MPI program **x**, and will run it on four machines in a cluster. Each machine will be running its own copy of **x**. Official MPI terminology refers to this as four **processes**. Now that multicore machines are commonplace, one might indeed run two or more cooperating MPI processes—where now we use the term *processes* in the real OS sense—on the same multicore machine. In this document, we will tend to refer to the various MPI processes as **nodes**, with an eye to the cluster setting.

Though the nodes are all running the same program, they will likely be working on different parts of the program's data. This is called the Single Program Multiple Data (SPMD) model. This is the typical approach, but there could be different programs running on different nodes. Most of the APIs involve a node sending information to, or receiving information from, other nodes.

7.1.3 Implementations

Two of the most popular implementations of MPI are MPICH and LAM. MPICH offers more tailoring to various networks and other platforms, while LAM runs on networks. Introductions to MPICH and LAM can be found, for example, at `http://heather.cs.ucdavis.edu/~matloff/MPI/NotesMPICH.NM.html` and `http://heather.cs.ucdavis.edu/~matloff/MPI/NotesLAM.NM.html`, respectively.

7.1.4 Performance Issues

Mere usage of a parallel language on a parallel platform does not guarantee a performance improvement over a serial version of your program. The central issue here is the overhead involved in internode communication.

As of 2008, Infiniband, one of the fastest cluster networks commercially available, has a **latency** of about 1.0-3.0 microseconds, meaning that it takes the first bit of a packet that long to get from one node on an Infiniband switch to another. Comparing that to the nanosecond time scale of CPU speeds, one can see that the communications overhead can destroy a program's performance. And Ethernet is quite a bit slower than Infiniband.

Note carefully that latency is a major problem even if the **bandwidth**—the number of bits per second which are sent—is high. For this reason, it is quite possible that your parallel program may actually run more slowly than its serial version.

Of course, if your platform is a shared-memory multiprocessor (especially a multicore one, where communication between cores is particularly fast) and you are running all your MPI processor on that machine, the problem is less severe.

7.2 Running Example

7.2.1 The Algorithm

The code implements the Dijkstra algorithm for finding the shortest paths in an undirected graph. Pseudocode for the algorithm is

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

At each iteration, the algorithm finds the closest vertex J to 0 among all those not yet processed, and then updates the list of minimum distances to each vertex from 0 by considering paths that go through J. Two obvious potential candidate part of the algorithm for parallelization are the “find J” and “for K” lines, and the above OpenMP code takes this approach.

7.2.2 The Code

```
// Dijkstra.c

// MPI example program:  Dijkstra shortest-path finder in a
// bidirectional graph; finds the shortest path from vertex 0 to all
// others

// command line arguments:  nv print dbg

// where:  nv is the size of the graph; print is 1 if graph and min
// distances are to be printed out, 0 otherwise; and dbg is 1 or 0, 1
// for debug

// node 0 will both participate in the computation and serve as a
// "manager"

#include <stdio.h>
```

```
#include <mpi.h>

#define MYMIN_MSG 0
#define OVRLMIN_MSG 1
#define COLLECT_MSG 2

// global variables (but of course not shared across nodes)

int nv,  // number of vertices
    *notdone, // vertices not checked yet
    nnodes,  // number of MPI nodes in the computation
    chunk,  // number of vertices handled by each node
    startv,endv,  // start, end vertices for this node
    me,  // my node number
    dbg;
unsigned largeint,  // max possible unsigned int
         mymin[2],  // mymin[0] is min for my chunk,
                    // mymin[1] is vertex which achieves that min
         othermin[2],  // othermin[0] is min over the other chunks
                       // (used by node 0 only)
                       // othermin[1] is vertex which achieves that min
         overallmin[2],  // overallmin[0] is current min over all nodes,
                         // overallmin[1] is vertex which achieves that min
         *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
                // ohd[i*nv+j]
         *mind;  // min distances found so far

double T1,T2;  // start and finish times

void init(int ac, char **av)
{  int i,j,tmp; unsigned u;
   nv = atoi(av[1]);
   dbg = atoi(av[3]);
   MPI_Init(&ac,&av);
   MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);
   chunk = nv/nnodes;
   startv = me * chunk;
   endv = startv + chunk - 1;
   u = -1;
   largeint = u >> 1;
   ohd = malloc(nv*nv*sizeof(int));
   mind = malloc(nv*sizeof(int));
   notdone = malloc(nv*sizeof(int));
   // random graph
   // note that this will be generated at all nodes; could generate just
   // at node 0 and then send to others, but faster this way
   srand(9999);
   for (i = 0; i < nv; i++)
      for (j = i; j < nv; j++)  {
         if (j == i) ohd[i*nv+i] = 0;
         else  {
            ohd[nv*i+j] = rand() % 20;
            ohd[nv*j+i] = ohd[nv*i+j];
         }
      }
   for (i = 0; i < nv; i++)  {
      notdone[i] = 1;
      mind[i] = largeint;
   }
   mind[0] = 0;
   while (dbg) ;  // stalling so can attach debugger
}

// finds closest to 0 among notdone, among startv through endv
void findmymin()
{  int i;
   mymin[0] = largeint;
   for (i = startv; i <= endv; i++)
      if (notdone[i] && mind[i] < mymin[0])  {
         mymin[0] = mind[i];
         mymin[1] = i;
      }
}

void findoverallmin()
{  int i;
   MPI_Status status;  // describes result of MPI_Recv() call
   // nodes other than 0 report their mins to node 0, which receives
   // them and updates its value for the global min
   if (me > 0)
      MPI_Send(mymin,2,MPI_INT,0,MYMIN_MSG,MPI_COMM_WORLD);
   else  {
      // check my own first
      overallmin[0] = mymin[0];
      overallmin[1] = mymin[1];
      // check the others
      for (i = 1; i < nnodes; i++)  {
         MPI_Recv(othermin,2,MPI_INT,i,MYMIN_MSG,MPI_COMM_WORLD,&status);
         if (othermin[0] < overallmin[0])  {
            overallmin[0] = othermin[0];
            overallmin[1] = othermin[1];
         }
      }
   }
}

void updatemymind()  // update my mind segment
{  // for each i in [startv,endv], ask whether a shorter path to i
   // exists, through mv
   int i, mv = overallmin[1];
   unsigned md = overallmin[0];
   for (i = startv; i <= endv; i++)
      if (md + ohd[mv*nv+i] < mind[i])
         mind[i] = md + ohd[mv*nv+i];
}

void disseminateoverallmin()
{  int i;
   MPI_Status status;
   if (me == 0)
      for (i = 1; i < nnodes; i++)
         MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD);
   else
      MPI_Recv(overallmin,2,MPI_INT,0,OVRLMIN_MSG,MPI_COMM_WORLD,&status);
}
```

```

void updateallmind()  // collects all the mind segments at node 0
{  int i;
   MPI_Status status;
   if (me > 0)
      MPI_Send(mind+startv,chunk,MPI_INT,0,COLLECT_MSG,MPI_COMM_WORLD);
   else
      for (i = 1; i < nnodes; i++)
         MPI_Recv(mind+i*chunk,chunk,MPI_INT,i,COLLECT_MSG,MPI_COMM_WORLD,
            &status);
}

void printmind()  // partly for debugging (call from GDB)
{  int i;
   printf("minimum distances:\n");
   for (i = 1; i < nv; i++)
      printf("%u\n",mind[i]);
}

void dowork()
{  int step,  // index for loop of nv steps
       i;
   if (me == 0) T1 = MPI_Wtime();
   for (step = 0; step < nv; step++)  {
      findmymin();
      findoverallmin();
      disseminateoverallmin();
      // mark new vertex as done
      notdone[overallmin[1]] = 0;
      updatemymind(startv,endv);
   }
   updateallmind();
   T2 = MPI_Wtime();
}

int main(int ac, char **av)
{  int i,j,print;
   init(ac,av);
   dowork();
   print = atoi(av[2]);
   if (print && me == 0)  {
      printf("graph weights:\n");
      for (i = 0; i < nv; i++)  {
         for (j = 0; j < nv; j++)
            printf("%u  ",ohd[nv*i+j]);
         printf("\n");
      }
      printmind();
   }
   if (me == 0) printf("time at node 0: %f\n",(float)(T2-T1));
   MPI_Finalize();
}

```

The various MPI functions will be explained in the next section.

7.2.3 Introduction to MPI APIs

7.2.3.1 MPI_Init() and MPI_Finalize()

These are required for starting and ending execution of an MPI program. Their actions may be implementation-dependent. For instance, if our platform is an Ethernet-based cluster , **MPI_Init()** will probably set up the TCP/IP sockets via which the various nodes communicate with each other. On an Infiniband-based cluster, connections in the special Infiniband network protocol will be established. On a shared-memory multiprocessor, an implementation of MPI that is tailored to that platform would take very different actions.

7.2.3.2 MPI_Comm_size() and MPI_Comm_rank()

In our function **init()** above, note the calls

```
MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
MPI_Comm_rank(MPI_COMM_WORLD,&me);
```

The first call determines how many nodes are participating in our computation, placing the result in our variable **nnodes**. Here **MPI_COMM_WORLD** is our node group, termed a **communicator** in MPI parlance. MPI allows the programmer to subdivide the nodes into groups, to facilitate performance and clarity of code. Note that for some operations, such as barriers, the only way to apply the operation to a proper subset of all nodes is to form a group. The totality of all groups is denoted by **MPI_COMM_WORLD**. In our program here, we are not subdividing into groups.

The second call determines this node's ID number, called its **rank**, within its group. As mentioned earlier, even though the nodes are all running the same program, they are typically working on different parts of the program's data. So, the program needs to be able to sense which node it is running on, so as to access the appropriate data. Here we record that information in our variable **me**.

7.2.3.3 MPI_Send()

To see how MPI's basic send function works, consider our line above,

```
MPI_Send(mymin,2,MPI_INT,0,MYMIN_MSG,MPI_COMM_WORLD);
```

Let's look at the arguments:

mymin: We are sending a set of bytes. This argument states the address at which these bytes begin.

2, MPI_INT: This says that our set of bytes to be sent consists of 2 objects of type **MPI_INT**. That means 8 bytes on 32-bit machines, so why not just collapse these two arguments to one, namely the number 8? Why did the designers of MPI bother to define data types? The answer is that we want to be able to run MPI on a heterogeneous set of machines, with MPI serving as the "broker" between them in case different architectures among those machines handle data differently.

First of all, there is the issue of **endianness**. Intel machines, for instance, are **little-endian**, which means that the least significant byte of a memory word has the smallest address among bytes of the word. Sun SPARC chips, on the other hand, are **big-endian**, with the opposite storage scheme. If our set of nodes included machines of both types, straight transmission of sequences of 8 bytes might mean that some of the machines literally receive the data backwards!

Secondly, these days 64-bit machines are becoming more and more common. Again, if our set of nodes were to include both 32-bit and 64-bit words, some major problems would occur if no conversion were done.

0: We are sending to node 0.

MYMIN_MSG: This is the message type, programmer-defined in our line

```
#define MYMIN_MSG 0
```

Receive calls, described in the next section, can ask to receive only messages of a certain type.

I_COMM_WORLD: This is the node group to which the message is to be sent. Above, where we said we are sending to node 0, we technically should say we are sending to node 0 within the group **MPI_COMM_WORLD**.

7.2.3.4 MPI_Recv()

Let's now look at the arguments for a basic receive:

```
MPI_Recv(othermin,2,MPI_INT,i,MYMIN_MSG,MPI_COMM_WORLD,&status);
```

othermin: The received message is to be placed at our location **othermin**.

2,MPI_INT: Two objects of **MPI_INT** type are to be received.

i: Receive only messages of from node **i**. If we did not care what node we received a message from, we could specify the value **MPI_ANY_SOURCE**.

MYMIN_MSG: Receive only messages of type **MYMIN_MSG**. If we did not care what type of message we received, we would specify the value **MPI_ANY_TAG**.

I_COMM_WORLD: Group name.

status: Recall our line

```
MPI_Status status;  // describes result of MPI_Recv() call
```

The type is an MPI **struct** containing information about the received message. Its primary fields of interest are **MPI_SOURCE**, which contains the identity of the sending node, and **MPI_TAG**, which contains the message type. These would be useful if the receive had been done with **MPI_ANY_SOURCE** or **MPI_ANY_TAG**; the status argument would then tell us which node sent the message and what type the message was.

7.3 Collective Communications

MPI features a number of **collective communication** capabilities, a number of which are used in the following refinement of our Dijkstra program:

7.3.1 Example

```
// Dijkstra.coll1.c

// MPI example program:  Dijkstra shortest-path finder in a
// bidirectional graph; finds the shortest path from vertex 0 to all
// others; this version uses collective communication

// command line arguments:  nv print dbg

// where:  nv is the size of the graph; print is 1 if graph and min
// distances are to be printed out, 0 otherwise; and dbg is 1 or 0, 1
// for debug

// node 0 will both participate in the computation and serve as a
// "manager"

#include <stdio.h>
#include <mpi.h>

// global variables (but of course not shared across nodes)

int nv,  // number of vertices
    *notdone, // vertices not checked yet
    nnodes,  // number of MPI nodes in the computation
    chunk,  // number of vertices handled by each node
    startv,endv,  // start, end vertices for this node
    me,  // my node number
    dbg;
unsigned largeint,  // max possible unsigned int
         mymin[2],  // mymin[0] is min for my chunk,
                    // mymin[1] is vertex which achieves that min
         overallmin[2],  // overallmin[0] is current min over all nodes,
                         // overallmin[1] is vertex which achieves that min
          *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
                 // ohd[i*nv+j]
          *mind;  // min distances found so far

double T1,T2;  // start and finish times

void init(int ac, char **av)
{  int i,j,tmp; unsigned u;
   nv = atoi(av[1]);
   dbg = atoi(av[3]);
   MPI_Init(&ac,&av);
   MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);
   chunk = nv/nnodes;
   startv = me * chunk;
   endv = startv + chunk - 1;
   u = -1;
   largeint = u >> 1;
   ohd = malloc(nv*nv*sizeof(int));
   mind = malloc(nv*sizeof(int));
   notdone = malloc(nv*sizeof(int));
   // random graph
   // note that this will be generated at all nodes; could generate just
   // at node 0 and then send to others, but faster this way
   srand(9999);
   for (i = 0; i < nv; i++)
      for (j = i; j < nv; j++)  {
         if (j == i) ohd[i*nv+i] = 0;
         else  {
            ohd[nv*i+j] = rand() % 20;
            ohd[nv*j+i] = ohd[nv*i+j];
         }
      }
   for (i = 0; i < nv; i++)  {
      notdone[i] = 1;
      mind[i] = largeint;
   }
   mind[0] = 0;
   while (dbg) ;  // stalling so can attach debugger
}

// finds closest to 0 among notdone, among startv through endv
void findmymin()
{  int i;
   mymin[0] = largeint;
   for (i = startv; i <= endv; i++)
      if (notdone[i] && mind[i] < mymin[0])  {
         mymin[0] = mind[i];
         mymin[1] = i;
      }
}

void updatemymind()  // update my mind segment
{  // for each i in [startv,endv], ask whether a shorter path to i
   // exists, through mv
   int i, mv = overallmin[1];
   unsigned md = overallmin[0];
   for (i = startv; i <= endv; i++)
      if (md + ohd[mv*nv+i] < mind[i])
         mind[i] = md + ohd[mv*nv+i];
}

void printmind()  // partly for debugging (call from GDB)
{  int i;
   printf("minimum distances:\n");
   for (i = 1; i < nv; i++)
      printf("%u\n",mind[i]);
}

void dowork()
{  int step,  // index for loop of nv steps
       i;
   if (me == 0) T1 = MPI_Wtime();
   for (step = 0; step < nv; step++)  {
      findmymin();
      MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
      MPI_Bcast(overallmin,1,MPI_2INT,0,MPI_COMM_WORLD);
      // mark new vertex as done
      notdone[overallmin[1]] = 0;
      updatemymind(startv,endv);
   }
   // now need to collect all the mind values from other nodes to node 0
   MPI_Gather(mind+startv,chunk,MPI_INT,mind,chunk,MPI_INT,0,MPI_COMM_WORLD);
   T2 = MPI_Wtime();
}

int main(int ac, char **av)
{  int i,j,print;
   init(ac,av);
   dowork();
   print = atoi(av[2]);
   if (print && me == 0)  {
      printf("graph weights:\n");
      for (i = 0; i < nv; i++)  {
         for (j = 0; j < nv; j++)
            printf("%u  ",ohd[nv*i+j]);
         printf("\n");
      }
      printmind();
   }
   if (me == 0) printf("time at node 0: %f\n",(float)(T2-T1));
   MPI_Finalize();
}
```

The new calls will be explained in the next section.

7.3.2 MPI_Bcast()

In our original Dijkstra example, we had a loop

```
for (i = 1; i < nnodes; i++)
   MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD);
```

in which node 0 sends to all other nodes. We can replace this by

```
MPI_Bcast(overallmin,2,MPI_INT,0,MPI_COMM_WORLD);
```

In English, this call would say,

> At this point all nodes participate in a broadcast operation, in which node 0 sends 2 objects of type **MPI_INT**. The source of the data will be located at address **overallmin** at node 0, and the other nodes will receive the data at a location of that name.

Note my word “participate” above. The name of the function is “broadcast,” which makes it sound like only node 0 executes this line of code, which is not the case; all the nodes in the group (in this case that means all nodes in our entire computation) execute this line. The only difference is the action; most nodes participate by receiving, while node 0 participates by sending.

Why might this be preferable to using an explicit loop? First, it would obviously be much clearer. That makes the program easier to write, easier to debug, and easier for others (and ourselves, later) to read.

But even more importantly, using the broadcast may improve performance. We may, for instance, be using an implementation of MPI which is tailored to the platform on which we are running MPI. If for instance we are running on a network designed for parallel computing, such as Myrinet or Infiniband, an optimized broadcast may achieve a much higher performance level than would simply a loop with individual send calls. On a shared-memory multiprocessor system, special machine instructions specific to that platform’s architecture can be exploited, as for instance IBM has done for its shared-memory machines. Even on an ordinary Ethernet, one could exploit Ethernet’s own broadcast mechanism, as had been done for PVM, a system like MPI (G. Davies and N. Matloff, Network-Specific Performance Enhancements for PVM, *Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing*, 1995, 205-210).

7.3.2.1 MPI_Reduce()/MPI_Allreduce()

Look at our call

```
MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
```

above. In English, this would say,

> At this point all nodes in this group participate in a "reduce" operation. The type of reduce operation is **MPI_MINLOC**, which means that the minimum value among the nodes will be computed, and the index attaining that minimum will be recorded as well. Each node contributes a value to be checked, and an associated index, from a location **mymin** in their programs; the type of the pair is **MPI_2INT**. The overall min value/index will be computed by combining all of these values at node 0, where they will be placed at a location **overallmin**.

MPI also includes a function **MPI_Allreduce()**, which does the same operation, except that instead of just depositing the result at one node, it does so at all nodes. So for instance our code above,

```
MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
MPI_Bcast(overallmin,1,MPI_2INT,0,MPI_COMM_WORLD);
```

could be replaced by

```
MPI_Allreduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,MPI_COMM_WORLD);
```

Again, these can be optimized for particular platforms.

7.3.2.2 MPI_Gather()/MPI_Allgather()

A classical approach to parallel computation is to first break the data for the application into chunks, then have each node work on its chunk, and then gather all the processed chunks together at some node. The MPI function **MPI_Gather()** does this.

In our program above, look at the line

```
MPI_Gather(mind+startv,chunk,MPI_INT,mind,chunk,MPI_INT,0,MPI_COMM_WORLD);
```

In English, this says,

> At this point all nodes participate in a gather operation, in which each node contributes data, consisting of **chunk** number of MPI integers, from a location **mind+startv** in its program. All that data is strung together and deposited at the location **mind** in the program running at node 0.

There is also **MPI_Allgather()**, which places the result at all nodes, not just one.

7.3.2.3 The MPI_Scatter()

This is the opposite of **MPI_Gather()**, i.e. it breaks long data into chunks which it parcels out to individual nodes.

Here is MPI code to count the number of edges in a directed graph. (A link from i to j does not necessarily imply one from j to i.) In the context here, **me** is the node's rank; **nv** is the number of vertices; **oh** is the one-hop distance matrix; and **nnodes** is the number of MPI processes. At the beginning only the process of rank 0 has a copy of **oh**, but it sends that matrix out in chunks to the other nodes, each of which stores its chunk in an array **ohchunk**.

```
MPI_Scatter(oh, nv*nv, MPI_INT, ohchunk, nv/nnodes, MPI_INT, 0,
MPI_COMM_WORLD);
mycount = 0;
for (i = 0; i < nv*nv/nnodes)
   if (ohchunk[i] != 0) mycount++;
MPI_Reduce(&mycount,&numedge,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if (me == 0) printf("there are %d edges\n",numedge);
```

7.3.2.4 The MPI_Barrier()

This implements a barrier for a given communicator. The name of the communicator is the sole argument for the function.

Explicit barriers are less common in message-passing programs than in the shared-memory world.

7.3.3 Creating Communicators

Again, a communicator is a subset (either proper or improper) of all of our nodes. MPI includes a number of functions for use in creating communicators. Some set up a virtual "topology" among the nodes.

For instance, many physics problems consist of solving differential equations in two- or three-dimensional space, via approximation on a grid of points. In two dimensions, groups may consists of rows in the grid.

We will not pursue this further here.

7.4 Buffering, Synchrony and Related Issues

As noted several times so far, interprocess communication in parallel systems can be quite expensive in terms of time delay. In this section we will consider some issues which can be extremely important in this regard.

7.4.1 Buffering, Etc.

To understand this point, first consider situations in which MPI is running on some network, under the TCP/IP protocol. Say an MPI program at node A is sending to one at node B.

It is extremely import to keep in mind the levels of abstraction here. The OS's TCP/IP stack is running at the Session, Transport and Network layers of the network. MPI—meaning the MPI internals—is running above the TCP/IP stack, in the Application layers at A and B. And the MPI user-written application could be considered to be running at a "Super-application" layer, since it calls the MPI internals. (From here on, we will refer to the MPI internals as simply "MPI.")

MPI at node A will have set up a TCP/IP socket to B during the user program's call to **MPI_Init()**. The other end of the socket will be a corresponding one at B. This setting up of this socket pair as establishing a **connection** between A and B. When node A calls **MPI_Send()**, MPI will write to the socket, and the TCP/IP stack will transmit that data to the TCP/IP socket at B. The TCP/IP stack at B will then send whatever bytes come in to MPI at B.

Now, it is important to keep in mind that in TCP/IP the totality of bytes sent by A to B during lifetime of the connection is considered one long message. So for instance if the MPI program at A calls **MPI_Send()** five times, the MPI internals will write to the socket five times, but the bytes from those five messages will not be perceived by the TCP/IP stack at B as five messages, but rather as just one long message (in fact, only part of one long message, since more may be yet to come).

MPI at B continually reads that "long message" and breaks it back into MPI messages, keeping them ready for calls to **MPI_Recv()** from the MPI application program at B. Note carefully that phrase, *keeping them ready*; it refers to the fact that the order in which the MPI application program requests those messages may be different from the order in which they arrive.

On the other hand, looking again at the TCP/IP level, even though all the bytes sent are considered one long message, it will physically be sent out in pieces. These pieces don't correspond to the pieces written to the socket, i.e. the MPI messages. Rather, the breaking into pieces is done for the purpose of **flow control**, meaning that the TCP/IP stack at A will not send data to the one at B if the OS at B has no room for it. The **buffer** space the OS at B has set up for receiving data is limited. As A is sending to B, the TCP layer at B is telling its counterpart at A when A is allowed to send more data.

Think of what happens the MPI application at B calls **MPI_Recv()**, requesting to receive from A, with a certain tag T. Say the first argument is named **x**, i.e. the data to be received is to be deposited at **x**. If MPI sees that it already has a message of tag T, it will have its **MPI_Recv()** function return the message to the caller, i.e. to the MPI application at B. **If no such message has arrived yet, MPI won't return to the caller yet, and thus the caller blocks.**

MPI_Send() can block too. If the platform and MPI implementation is that of the TCP/IP network context described above, then the send call will return when its call to the OS' **write()** (or equivalent, depending on OS) returns, but that could be delayed if the OS' buffer space is full. On the other hand, another implementation could require a positive response from B before allowing the send call to return.

Note that buffering slows everything down. In our TCP scenario above, **MPI_Recv()** at B must copy messages from the OS' buffer space to the MPI application program's program variables, e.g. **x** above. This is definitely a blow to performance. That in fact is why networks developed specially for parallel processing typically include mechanisms to avoid the copying. Infiniband, for example, has a Remote Direct Memory Access capability, meaning that A can write directly to **x** at B. Of course, if our implementation uses **synchronous** communication, with A's send call not returning until A gets a response from B, we must wait even longer.

Technically, the MPI standard states that **MPI_Send(x,...)** will return only when it is safe for the application program to write over the array which it is using to store its message, i.e. **x**. As we have seen, there are various ways to implement this, with performance implications. Similarly, **MPI_Recv(y,...)** will return only when it is safe to read **y**.

7.4.2 Safety

With **synchronous** communication, deadlock is a real risk. Say A wants to send two messages to B, of types U and V, but that B wants to receive V first. Then A won't even get to send V, because in preparing to send U it must wait for a notice from B that B wants to read U—a notice which will never come, because B sends such a notice for V first. This would not occur if the communication were asynchronous.

But beyond formal deadlock, programs can fail in other ways, even with buffering, as buffer space is always by nature finite. A program can fail if it runs out of buffer space, either at the sender or the receiver. See `www.llnl.gov/computing/tutorials/mpi_performance/samples/unsafe.c` for an example of a test program which demonstrates this on a certain platform, by deliberating overwhelming the buffers at the receiver.

In MPI terminology, asynchronous communication is considered **unsafe**. The program may run fine on most systems, as most systems are buffered, but fail on some systems. Of course, as long as you know your program won't be run in nonbuffered settings, it's fine, and since there is potentially such a performance penalty for doing things synchronously, most people are willing to go ahead with their "unsafe" code.

7.4.3 Living Dangerously

If one is sure that there will be no problems of buffer overflow and so on, one can use variant send and receive calls provided by MPI, such as **MPI_Isend()** and **MPI_Irecv()**. The key difference between them and **MPI_Send()** and **MPI_Recv()** is that they return immediately, and thus are termed **nonblocking**. Your code can go on and do other things, not having to wait.

This does mean that at A you cannot touch the data you are sending until you determine that it has either been buffered somewhere or has reached **x** at B. Similarly, at B you can't use the data at **x** until you determine that it has arrived. Such determinations can be made via **MPI_Wait()**. In other words, you can do your send or receive, then perform some other computations for a while, and then call **MPI_Wait()** to determine whether you can go on. Or you can call **MPI_Probe()** to ask whether the operation has completed yet.

7.4.4 Safe Exchange Operations

In many applications A and B are swapping data, so both are sending and both are receiving. This too can lead to deadlock. An obvious solution would be, for instance, to have the lower-rank node send first and the higher-rank node receive first.

But a more convenient, safer and possibly faster alternative would be to use MPI's **MPI_Sendrecv()** function. Its prototype is

```
intMPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
   int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
   MPI_Status *status)
```

Note that the sent and received messages can be of different lengths and can use different tags.

7.5 Use of MPI from Other Languages

MPI is a vehicle for parallelizing C/C++, but some clever people have extended the concept to other languages, such as the cases of Python and R that we treat here.

7.5.1 Python: pyMPI

(**Important note**: As of April 2010, a much more widely used Python/MPI interface is MPI4Py. It works similarly to what is described here.)

A number of interfaces of Python to MPI have been developed.[1] A well-known example is pyMPI, developed by a PhD graduate in computer science in UCD, Patrick Miller.

One writes one's pyMPI code, say in **x.py**, by calling pyMPI versions of the usual MPI routines. To run the code, one then runs MPI on the program **pyMPI** with **x.py** as a command-line argument.

[1]If you are not familiar with Python, I have a quick tutorial at `http://heather.cs.ucdavis.edu/~matloff/python.html`.

Python is a very elegant language, and pyMPI does a nice job of elegantly interfacing to MPI. Following is a rendition of Quicksort in pyMPI. Don't worry if you haven't worked in Python before; the "non-C-like" Python constructs are explained in comments at the end of the code.

```
# a type of quicksort; break array x (actually a Python "list") into
# p quicksort-style piles, based # on comparison with the first p-1
# elements of x, where p is the number # of MPI nodes; the nodes sort
# their piles, then return them to node 0, # which strings them all
# together into the final sorted array

import mpi  # load pyMPI module

# makes npls quicksort-style piles
def makepiles(x,npls):
   pivot = x[:npls]  # we'll use the first npls elements of x as pivots,
                     # i.e. we'll compare all other elements of x to these
   pivot.sort()  # sort() is a member function of the Python list class
   pls = []  # initialize piles list to empty
   lp = len(pivot)  # length of the pivot array
   # pls will be a list of lists, with the i-th list in pls storing the
   # i-th pile; the i-th pile will start with ID i (to enable
   # identification later on) and pivot[i]
   for i in range(lp):  # i = 0,1,...lp-1
      pls.append([i,pivot[i]])  # build up array via append() member function
   pls.append([lp])
   for xi in x[npls:]:  # now place each element in the rest of x into
                        # its proper pile
      for j in range(lp):  # j = 0,1,...,lp-1
         if xi <= pivot[j]:
            pls[j].append(xi)
            break
         elif j == lp-1: pls[lp].append(xi)
   return pls

def main():
   if mpi.rank == 0:  # analog of calling MPI_Rank()
      x = [12,5,13,61,9,6,20,1]  # small test case
      # divide x into piles to be disbursed to the various nodes
      pls = makepiles(x,mpi.size)
   else:  # all other nodes set their x and pls to empty
      x = []
      pls = []
   mychunk = mpi.scatter(pls)  # node 0 (not an explicit argument) disburses
                               # pls to the nodes, each of which receives
                               # its chunk in its mychunk
   newchunk = []  # will become sorted version of mychunk
   for pile in mychunk:
      # I need to sort my chunk but most remove the ID first
      plnum = pile.pop(0)  # ID
      pile.sort()
      # restore ID
      newchunk.append([plnum]+pile)  # the + is array concatenation
   # now everyone sends their newchunk lists, which node 0 (again an
   # implied argument) gathers together into haveitall
   haveitall = mpi.gather(newchunk)
   if mpi.rank == 0:
      haveitall.sort()
      # string all the piles together
      sortedx = [z for q in haveitall for z in q[1:]]
      print sortedx

# common idiom for launching a Python program
if __name__ == '__main__': main()
```

Some examples of use of other MPI functions:

```
mpi.send(mesgstring,destnodenumber)
(message,status) = mpi.recv()  # receive from anyone
print message
(message,status) = mpi.recv(3)  # receive only from node 3
(message,status) = mpi.recv(3,ZMSG)  # receive only message type ZMSG,
                                     # only from node 3
(message,status) = mpi.recv(tag=ZMSG)  # receive from anyone, but
                                       # only message type ZMSG
```

7.5.2 R

7.5.2.1 Rmpi

The Rmpi package provides an interface from R to MPI, much like that of pyMPI.[2]

So, we run Rmpi on top of MPI. Even nicer, we can run the Snow package on top of Rmpi! Snow provides a higher-level interface, which is very convenient.

Installation:

Say you want to install in the directory **/a/b/c/**. The easiest way to do so is

```
> install.packages("Rmpi","/a/b/c/")
```

This will install Rmpi in the directory **/a/b/c/Rmpi**.

You'll need to arrange for the directory **/a/b/c** (not **/a/b/c/Rmpi**) to be added to your R library search path. I recommend placing a line

```
.libPaths("/a/b/c/")
```

in a file **.Rprofile** in your home directory.

[2]R is a widely-used language for statistics/data. I have a programmer's tutorial for it at `http://heather.cs.ucdavis.edu/~matloff/R/RProg.pdf`.

Usage:

Fire up MPI, and then in R load in Rmpi, by typing

```
> library(Rmpi)
```

Then start Rmpi:

```
> mpi.spawn.Rslaves()
```

This will start R on all machines in the group you started MPI on. Optionally, you can specify fewer machines via the named argument **nslaves**.

The first time you do this, try this test:

```
mpi.remote.exec(paste("I am",mpi.comm.rank(),"of",mpi.comm.size()))
```

The available functions are similar to those of pyMPI, such as

- **mpi.comm.size():**
  Returns the number of MPI processes, including the master that spawned the other processes.
- **mpi.comm.rank():**
  Returns the rank of the process that executes it.
- **mpi.send()**, mpi.recv():
  The usual send/receive operations.
- **mpi.bcast(), mpi.scatter(), mpi.gather():**
  The usual broadcast, scatter and gather operations.
- Etc.

Details are available at:

- `http://cran.r-project.org/web/packages/Rmpi/index.html`
  Site for download of package and manual.
- `http://ace.acadiau.ca/math/ACMMaC/Rmpi/sample.html`
  Nice tutorial.

But we forego details here, as Snow provides a nicer programmer interface, to be described next.

7.5.2.2 The R snow Package

Snow runs on top of Rmpi (or directly via sockets), allowing the programmer to more conveniently express the parallel disposition of work.

For instance, just as the ordinary R function **apply()** applies the same function to all rows of a matrix, the Snow function **parApply()** does that in parallel, across multiple machines; different machines will work on different rows.

Installation:

Follow the same pattern as described above for Rmpi. If you plan to have Snow run on top of Rmpi, you'll of course need the latter too.

Usage:

Make sure Snow is in your library path (see material on Rmpi above).

Load Snow:

```
> library(snow)
```

One then sets up a cluster, by calling the Snow function **makeCluster()**. The named argument **type** of that function indicates the networking platform, e.g. "MPI," "PVM" or "SOCK." The last indicates that you wish Snow to run on TCP/IP sockets that it creates itself, rather than going through MPI.

It is generally preferable to use MPI for Snow. This provides more flexibility, as one's code could include calls to both Snow functions and MPI (i.e. Rmpi) functions. In the examples here, I used "SOCK," on machines named **pc48** and **pc49**, setting up the cluster this way:

```
> cls <- makeCluster(type="SOCK",c("pc48","pc49"))
```

For MPI or PVM, one specifies the number of nodes to create, rather than specifying the nodes themselves.

Note that the above R code sets up worker nodes at the machines named **pc48** and **pc49**; these are in addition to the master node, which is the machine on which that R code is executed

There are various other optional arguments. One you may find useful is **outfile**, which records the result of the call in the file **outfile**. This can be helpful if the call fails.

Let's look at a simple example of multiplication of a vector by a matrix. We set up a test matrix:

```
> a <- matrix(c(1,2,3,4,5,6,7,8,9,10,11,12),nrow=6)
> a
     [,1] [,2]
```

```
[1,]    1    7
[2,]    2    8
[3,]    3    9
[4,]    4   10
[5,]    5   11
[6,]    6   12
```

We will multiply the vector $(1,1)^T$ (T meaning transpose) by our matrix **a**, by defining a dot product function:

```
> dot <- function(x,y) {return(x%*%y)}
```

Let's test it using the ordinary **apply()**:

```
> apply(a,1,dot,c(1,1))
[1]  8 10 12 14 16 18
```

To review your R, note that this applies the function **dot()** to each row (indicated by the 1, with 2 meaning column) of **a** playing the role of the first argument to **dot()**, and with c(1,1) playing the role of the second argument.

Now let's do this in parallel, across our two machines in our cluster **cls**:

```
> parApply(cls,a,1,dot,c(1,1))
[1]  8 10 12 14 16 18
```

The function **clusterCall(cls,f,args)** applies the given function **f()** at each worker node in the cluster **cls**, using the arguments provided in **args**.

The function **clusterExport(cls,varlist)** copies the variables in the list **varlist** to each worker in the cluster **cls**. You can use this to avoid constant shipping of large data sets from the master to the workers; you just do so once, using **clusterExport()** on the corresponding variables, and then access those variables as global. For instance:

```
> z <- function() return(x)
> x <- 5
> y <- 12
> clusterExport(cls,list("x","y"))
> clusterCall(cls,z)
[[1]]
[1] 5

[[2]]
[1] 5
```

The function **clusterEvalQ(cls,expression)** runs **expression** at each worker node in **cls**. Continuing the above example, we have

```
> clusterEvalQ(cls,x <- x+1)
[[1]]
[1] 6

[[2]]
[1] 6

> clusterCall(cls,z)
[[1]]
[1] 6

[[2]]
[1] 6

> x
[1] 5
```

Note that **x** still has its original version back at the master.

The function **clusterApply(cls,individualargs,f,commonargsgohere)** runs **f()** at each worker node in **cls**, with arguments as follows. The first argument to **f()** for worker i is the ith element of the list **individualargs**, i.e. **individualargs[[i]]**, and optionally one can give additional arguments for **f()** following **f()** in the argument list for **clusterApply()**.

Here for instance is how we can assign an ID to each worker node, like MPI **rank**:[3]

```
> myid <- 0
> clusterExport(cls,"myid")
> setid <- function(i) {myid <<- i}  # note superassignment operator
> clusterApply(cls,1:2,setid)
[[1]]
[1] 1

[[2]]
[1] 2

> clusterCall(cls,function() {return(myid)})
[[1]]
[1] 1

[[2]]
[1] 2
```

Don't forget to stop your clusters before exiting R, by calling **stopCluster()clustername**.

There are various other useful Snow functions. See the user's manual for details.

[3]I don't see a provision in Snow itself that does this.

To learn more about Snow:

I recommend the following Web pages:

- `http://cran.cnr.berkeley.edu/web/packages/snow/index.html`
  CRAN page for Snow; the package and the manual are here.
- `http://www.bepress.com/cgi/viewcontent.cgi?article=1016&context=uwbiostat`
  A research paper.
- `http://www.cs.uiowa.edu/~luke/R/cluster/cluster.html`
  Brief intro by the author.
- `http://www.sfu.ca/~sblay/R/snow.html#clusterCall`
  Examples, short but useful.

Chapter 8

Introduction to Parallel Matrix Operations

8.1 Overview

In the early days parallel processing was mostly used in physics problems. Typical problems of interest would be grid computations such as the heat equation, matrix multiplication, matrix inversion (or equivalent operations) and so on. These matrices are not those little 3x3 things you worked with in your linear algebra class. In parallel processing applications of matrix algebra, our matrices can have thousands of rows and columns, or even larger.

The range of applications of parallel processing is of course far broader today. In many of these applications, problems which at first glance seem not to involve matrices, actually do have matrix solutions. An example in graph theory is the following.

Let n denote the number of vertices in the graph. Define the graph's **adjacency matrix** A to be the n x n matrix whose element (i,j) is equal to 1 if there is an edge connecting vertices i an j (i.e. i and j are "adjacent"), and 0 otherwise. The corresponding **reachability matrix** R has its (i,j) element equal to 1 if there is some path from i to j, and 0 otherwise.

One can prove that

$$R = b[(I + A)^{n-1}], \tag{8.1}$$

where I is the identity matrix and the function b() ('b' for "boolean") is applied elementwise to its matrix argument, replacing each nonzero element by 1 while leaving the elements which are 0 unchanged. The graph is connected if and only if all elements of R are 1s.

So, the original graph connectivity problem reduces to a matrix problem.

8.2 Partitioned Matrices

Parallel processing of course relies on finding a way to partition the work to be done. In the matrix algorithm case, this is often done by dividing a matrix into blocks (often called **tiles** these days).

For example, let

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{8.2}$$

and

$$B = \begin{pmatrix} 0 & 2 & 5 \\ 0 & 9 & 10 \\ 1 & 1 & 2 \end{pmatrix}, \tag{8.3}$$

so that

$$C = AB = \begin{pmatrix} 12 & 59 & 79 \\ 6 & 33 & 42 \\ 2 & 82 & 104 \end{pmatrix}. \tag{8.4}$$

We could partition A as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \tag{8.5}$$

where

$$A_{00} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{8.6}$$

$$A_{01} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{8.7}$$

$$A_{10} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{8.8}$$

and

$$A_{11} = \left(\begin{array}{c} 2 \end{array} \right). \tag{8.9}$$

Similarly we would partition B and C into blocks of the same size as in A,

$$B = \left(\begin{array}{cc} B_{00} & B_{01} \\ B_{10} & B_{11} \end{array} \right) \tag{8.10}$$

and

$$C = \left(\begin{array}{cc} C_{00} & C_{01} \\ C_{10} & C_{11} \end{array} \right), \tag{8.11}$$

so that for example

$$B_{10} = \left(\begin{array}{cc} 1 & 1 \end{array} \right). \tag{8.12}$$

The key point is that multiplication still works if we pretend that those submatrices are numbers. For example, pretending like that would give the relation

$$C_{00} = A_{00}B_{00} + A_{01}B_{10}, \tag{8.13}$$

which the reader should verify really is correct as matrices, i.e. the computation on the right side really does yield a matrix equal to C_{00}.

8.3 Matrix Multiplication

Let's suppose for the sake of simplicity that each of the matrices is of dimensions nxn. Let p denote the number of "processes," such as shared-memory threads or message-passing nodes.

We assume that the matrices are **dense**, meaning that most of their entries are nonzero. This is in contrast to **sparse** matrices, with many zeros. For instance, in **tridiagonal** matrices, in which the only nonzero elements are either on the diagonal or on subdiagonals just below or above the diagonal, and all other elements are guaranteed to be 0. Or we might just know that most elements are zeros but have no guarantee as to where they are; here we might have a system of pointers to get from one nonzero element to another. Clearly we would use differents type of algorithms for sparse matrices than for dense ones.

8.3.1 Message-Passing Case

For concreteness here and in other sections below on message passing, assume we are using MPI.

The obvious plan of attack here is to break the matrices into blocks, and then assign different blocks to different MPI nodes. Assume that $\sqrt{p}$ evenly divides n, and partition each matrix into submatrices of size $n/\sqrt{p}$ x $n/\sqrt{p}$. In other words, each matrix will be divided into m rows and m columns of blocks, where $m = n/\sqrt{p}$.

One of the conditions assumed here is that the matrices A and B are stored in a distributed manner across the nodes. This situation could arise for several reasons:

- The application is such that it is natural for each node to possess only part of A and B.
- One node, say node 0, originally contains all of A and B, but in order to conserve communication time, it sends each node only parts of those matrices.
- The entire matrix would not fit in the available memory at the individual nodes.

As you'll see, the algorithms then have the nodes passing blocks among themselves.

8.3.1.1 Fox's Algorithm

Consider the node that has the responsibility of calculating block (i,j) of the product C, which it calculates as

$$A_{i0}B_{0j} + A_{i1}B_{1j} + ... + A_{ii}B_{ij} + ... + A_{i,m-1}B_{m-1,j} \tag{8.14}$$

Rearrange this as

$$A_{ii}B_{ij} + A_{i,i+1}B_{,i+1j} + ... + A_{i,m-1}B_{m-1,j} + A_{i0}B_{0j} + A_{i1}B_{1j} + ... + A_{i,i-1}B_{i-1,j} \tag{8.15}$$

Written more compactly, this is

$$\sum_{k=0}^{m-1} A_{i,(i+k) mod\ m} B_{(i+k) mod\ m,j} \tag{8.16}$$

In other words, start with the A_{ii} term, then go down column i of A, wrapping back up to the top when you reach the bottom. The order of summation in this rearrangement will be the actual order of computation.

The algorithm is then as follows. The node which is handling the computation of C_{ij} does this (in parallel with the other nodes which are working with their own values of i and j):

```
iup  = i+1 mod m;
idown = i-1 mod m;
for (k = 0; k < m; k++) {
    km = (i+k) mod m;
    broadcast(A[i,km]) to all nodes handling row i of C;
    C[i,j] = C[i,j] + A[i,km]*B[km,j]
    send B[km,j] to the node handling C[idown,j]
    receive new B[km+1 mod m,j] from the node handling C[iup,j]
}
```

This is Fox's algorithm. Cannon's algorithm is similar, except that it does cyclical rotation in both rows and columns, compared to Fox's rotation only in columns but broadcast within rows.

The algorithm can be adapted in the obvious way to nonsquare matrices, etc.

8.3.1.2 Performance Issues

Note that in MPI we would probably want to implement this algorithm using communicators. For example, this would make broadcasting within a block row more convenient and efficient.

Note too that there is a lot of opportunity here to overlap computation and communication, which is the best way to solve the communication problem. For instance, we can do the broadcast above at the same time as we do the computation.

Obviously this algorithm is best suited to settings in which we have PEs in a mesh topology. This includes hypercubes, though one needs to be a little more careful about communications costs there.

8.3.2 Shared-Memory Case

8.3.2.1 OpenMP

Since a matrix multiplication in serial form consists of nested loops, a natural way to parallelize the operation in OpenMP is through the **for** pragma, e.g.

```
#pragma omp parallel for
for (i = 0; i < ncolsa; i++)
   for (j = 0; i < nrowsb; j++) {
      sum = 0;
      for (k = 0; i < ncolsa; i++)
         sum += a[i][k] * b[k][j];
   }
```

This would parallelize the outer loop, and we could do so at deeper nesting levels if profitable.

8.3.2.2 CUDA

Given that CUDA tends to work better if we use a large number of threads, a natural choice is for each thread to compute one element of the product, like this:

```
__global__ void matmul(float *ma,float *mb,float *mc,int nrowsa,int ncolsa,int ncolsb)
{  int k;
   sum = 0;
   for (k = 0; i < ncolsa; i++)
      sum += a[i*ncolsa+k] * b[k*ncols+j];
}
```

This should produce a good speedup. But we can do even better. Prof. Richard Edgar has tried making use of shared memory (`http://astro.pas.rochester.edu/~aquillen/gpuworkshop/AdvancedCUDA.pdf`):

```
__global__ void MultiplyOptimise(const float *A, const float *B, float *C) {
  // Extract block and thread numbers
  int bx = blockIdx.x; int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  // Index of first A sub-matrix processed by this block
  int aBegin = dc_wA * BLOCK_SIZE * by;
  // Index of last A sub-matrix
  int aEnd = aBegin + dc_wA - 1;
  // Stepsize of A sub-matrices
  int aStep = BLOCK_SIZE;
  // Index of first B sub-matrix
  // processed by this block
  int bBegin = BLOCK_SIZE * bx;
  // Stepsize for B sub-matrices
  int bStep = BLOCK_SIZE * dc_wB;
  // Accumulator for this thread
  float Csub = 0;
  for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b+= bStep) {
     // Shared memory for sub-matrices
     __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
     __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
     // Load matrices from global memory into shared memory
     // Each thread loads one element of each sub-matrix
     As[ty][tx] = A[a + (dc_wA * ty) + tx];
     Bs[ty][tx] = B[b + (dc_wB * ty) + tx];
     // Synchronise to make sure load is complete
     __syncthreads();
     // Perform multiplication on sub-matrices
     // Each thread computes one element of the C sub-matrix
     for( int k = 0; k < BLOCK_SIZE; k++ ) {
        Csub += As[ty][k] * Bs[k][tx];
```

```
      }
     // Synchronise again
     __syncthreads();
    }
    // Write the C sub-matrix back to global memory
    // Each thread writes one element
    int c = (dc_wB * BLOCK_SIZE * by) + (BLOCK_SIZE*bx);
    C[c + (dc_wB*ty) + tx] = Csub;
}
```

Here are the relevant portions of the calling code, including global variables giving the number of columns ("width") of the multiplier matrix and the number of rows ("height") of the multiplicand:

```
#define BLOCK_SIZE 16
...
__constant__ int dc_wA;
__constant__ int dc_wB;
...
// Sizes must be multiples of BLOCK_SIZE
dim3 threads(BLOCK_SIZE,BLOCK_SIZE);
dim3 grid(wB/BLOCK_SIZE,hA/BLOCK_SIZE);
MultiplySimple<<<grid,threads>>>(d_A, d_B, d_C);
...
```

(Note the alternative way to configure threads, using the functions **threads()** and **grid**().)

Here the the term "block" in the defined value **BLOCK_SIZE** refers both to blocks of threads and the partitioning of matrices. In other words, a thread block consists of 256 threads, to be thought of as a 16x16 "array" of threads, and each matrix is partitioned into submatrices of size 16x16.

In addition, in terms of grid configuration, there is again a one-to-one correspondence between thread blocks and submatrices. Each submatrix of the product matrix C will correspond to, and will be computed by, one block in the grid.

We are computing the matrix product C = AB. Denote the elements of A by a_{ij} for the element in row i, column j, and do the same for B and C. Row-major storage is used.

Each thread will compute one element of C, i.e. one c_{ij}. It will do so in the usual way, by multiplying column j of B by row i of A. However, the key issue is how this is done in concert with the other threads, and the timing of what portions of A and B are in shared memory at various times.

Concerning the latter, note the code

```
for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b+= bStep) {
   // Shared memory for sub-matrices
   __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
   // Load matrices from global memory into shared memory
   // Each thread loads one element of each sub-matrix
   As[ty][tx] = A[a + (dc_wA * ty) + tx];
   Bs[ty][tx] = B[b + (dc_wB * ty) + tx];
```

Here we loop across a row of submatrices of A, and a column of submatrices of B, calculating one submatrix of C. In each iteration of the loop, we bring into shared memory a new submatrix of A and a new one of B. Note how even this copying from device global memory to device shared memory is shared among the threads.

As an example, suppose

$$A = \left(\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) \tag{8.17}$$

and

$$B = \left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{array} \right) \tag{8.18}$$

Further suppose that **BLOCK_SIZE** is 2. That's too small for good efficiency—giving only four threads per block rather than 256—but it's good for the purposes of illustration.

Let's see what happens when we compute C_{11}, the 2x2 submatrix of C's upper-left corner. Due to the fact that partitioned matrices multiply "just like numbers," we have

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} && (8.19) \\ &= \left(\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \right) \left(\begin{array}{cc} 1 & 2 \\ 5 & 6 \end{array} \right) + ... && (8.20) \end{aligned}$$

Now, all this will be handled by thread block number (0,0), i.e. the block whose X and Y "coordinates" are both 0. In the first iteration of the loop, A_{11} and B_{11} are copied to shared memory for that block, then in the next iteration, A_{12} and B_{21} are brought in, and so on.

Consider what is happening with thread number (1,0) within that block. Remember, its ultimate goal is to compute c_{21} (adjusting for the fact that in math, matrix subscripts start at 1). In the first iteration, this thread is computing

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \end{pmatrix} = 11 \tag{8.21}$$

It saves that 11 in its running total **Csub**, eventually writing it to the corresponding element of C:

```
int c = (dc_wB * BLOCK_SIZE * by) + (BLOCK_SIZE*bx);
C[c + (dc_wB*ty) + tx] = Csub;
```

Professor Edgar found that use of shared device memory resulted a huge improvement, extending the original speedup of 20X to 500X!

8.3.3 Finding Powers of Matrices

In Section (8.1), we saw a special case of matrix multiplication, powers, so that in our context here we have A = B. There are some small improvements that we could make in our algorithm for this case, but also there is something big.

Suppose for instance we need to find A^{32}. We could apply the above algorithm 31 times. But a much faster approach would be to first calculate A^2, then square that result to get A^4, then square it to get A^8 and so on. That would get us A^{32} by applying the algorithm in Section 8.3.1.1 only five times, instead of 31.

8.4 Solving Systems of Linear Equations

Suppose we have a system of equations

$$a_{i0}x_0 + ... + a_{i,n-1}x_{n-1} = b_i, i = 0, 1, ..., n-1, \tag{8.22}$$

where the x_i are the unknowns to be solved for.

As you know, this system can be represented compactly as

$$AX = B, \tag{8.23}$$

where A is nxn and X and B are nx1.

In theory, this system could be solved by finding A^{-1} and left-multiplying by it on both sides of (8.23). However, in practice, this is never done, due to potential problems with numerical stability, etc. There are many other ways (some of which amount to finding A^{-1} indirectly).

8.4.1 Gaussian Elimination

You learned this in high school, and in your linear algebra course. Form the n x (n+1) matrix C = (A | B) by appending the column vector B to the right of A. Then we work on the rows of C, with the pseudocode for the sequential case in the most basic form being

```
for ii = 0 to n-1
   divide row ii by c[i][i]
   for r = ii+1 to n-1  // vacuous if r = n-1
      replace row r by row r - c[r][ii] times row ii
set new B to be column n-1 of C
```

This transforms C to upper triangular form, i.e. all the elements c_{ij} with $i > j$ are 0. Also, all diagonal elements are equal to 1. This corresponds to a new set of equations,

$$\begin{aligned}
c_{00}x_0 + c_{11}x_1 + c_{22}x_2 + ... + c_{0,n-1}x_{n-1} &= b_0 \\
c_{11}x_1 + c_{22}x_2 + ... + c_{1,n-1}x_{n-1} &= b_1 \\
c_{22}x_2 + ... + c_{2,n-1}x_{n-1} &= b_2 \\
... & \\
c_{n-1,n-1}x_{n-1} &= b_{n-1}
\end{aligned}$$

We then find the x_i via **back substitution**:

```
x[n-1] = b[n-1] / c[n-1,n-1]
for i = n-2 downto 0
   x[i] = (b[i] - c[i][n-1] * x[n-1] - ... - c[i][i+1] * x[i+1]) / c[i][i]
```

An obvious parallelization of this algorithm would be to assign each node one contiguous group of rows. Then each node would do

```
for ii = 0 to n-1
   if ii is in my group of rows
      pivot = c[i][i]
      divide row ii by pivot
      broadcast row ii
   else receive row ii
   for r = ii+1 to n-1 in my group
      subtract c[r][ii] times row ii from row r
set new B to be column n-1 of C
```

One problem with this is that in the outer loop, when ii gets past a given node's group of column indices, that node becomes idle. This can be solved by giving each node several groups of rows, in cyclic order. For example, say we have four nodes. Then node 0 could take rows 0-99, 400-499, 800-899 and so on, node 1 would take rows 100-199, 500-599 etc.

8.4.2 Iterative Methods

8.4.2.1 The Jacobi Algorithm

One can rewrite (8.22) as

$$x_i = \frac{1}{a_{ii}}[b_i - (a_{i0}x_0 + ... + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} + ... + a_{i,n-1}x_{n-1})], i = 0, 1, ..., n-1. \tag{8.24}$$

This suggests a natural iterative algorithm for solving the equations. We start with our guess being, say, $x_i = b_i$ for all i. At our kth iteration, we find our (k+1)st guess by plugging in our kth guess into the right-hand side of (8.24). We keep iterating until the difference between successive guesses is small enough to indicate convergence.

Parallelization of this algorithm is easy: Just assign each node to handle a block of X. Note that this means that each node must send its portion of the new X after every iteration.

8.4.2.2 The Gauss-Seidel Algorithm

This is a variant on the Jacobi algorithm, motivated by the following observation: In a sequential implementation of (8.24), when we get to x_i, we already know the new values of $x_0, x_1, ..., x_{i-1}$. Intuitively, we can speed up convergence of our algorithm by using those new values instead of the old ones.

In the parallel case, the easiest way to implement would be that, although we still assign each node to groups of the x_i, we would do this in a cyclic order as in Section 8.4.1.

8.5 The Shared-Memory Case

We can use all of these algorithms in the shared memory setting, with obvious modifications, a major one being that we remove the code that does send, receive and broadcast, as well as code (e.g. in the matrix multiplication algorithms) that moves data.

However: Keep in mind that in shared-memory settings, we are in effect doing send, receive and broadcast anyway. Every one thread does a write, it means that at some later time some other thread will try to read that item, which will mean that the latest copy of that item will be send to this second thread. That sending will be done either as a cache coherency transation in the case of shared-memory hardware, or as a similar page transation in the software distributed shared memory case.

That in turn means that we have to do our best to avoid false sharing. For instance, in the Gaussian elimination case, we have to make sure that the total number of bytes in a group of rows is a multiple of cache line size if we have shared-memory hardware, or a multiple of page size in the software case. We must also make sure that each group of rows begins at cache line/page boundaries. This is not hard, since our matrices will be stored in memory allocated by **malloc()** and the like. We may need to add some padding.

Chapter 9

Parallel Combinitorial Algorithms

9.1 Overview

In Chapter 1, we saw Dijkstra's algorithm for finding the shortest path in a graph. In Chapter 8, we saw an algorithm for finding bridges within a graph. Both of these are **combinatorial search algorithms**. Such algorithms generally have exponential time complexity, and thus are natural candidates for parallel computation. This chapter will present a few more examples.

9.2 The 8 Queens Problem

A famous example is the 8 Queens Problem, in which one wishes to place eight queens on a standard 8x8 chessboard in such a way that no queen is attacking any other. (The generalization, of course, would involve n queens on an nxn board.) Suppose our goal is to find all possible solutions.

To start a solution to this problem, we first note in any solution will have the property that no row will contain more than one queen. This suggests building up a solution row by row: Suppose we have successfully placed queens so far in rows 0, 1, ..., k-1 (row 0 being the top row of the board). Where can we place a queen in row k? Well, since we cannot use any column already occupied by the preceding k queens, that means we have a choice of 8-k columns. But even among those k columns, there will be j of them, for some $0 \leq j \leq 8-k$ that are in the diagonal attack path of some preceding queen. Then we can extend our tentative k-row solution to 8-k-j new (k+1)-row solutions.

We will define our solution here for the shared-memory paradigm, though it would be easy to change this for the message-passing paradigm.[1] Define

[1]The main point would be to change linked lists and pointers to arrays and array indices.

```
struct TentSoln {
   int RowsSoFar;
   int Cols[8];
   struct TentSoln *Next;
}
```

Each such **struct** contains a partial solution, up through row number **RowsSoFar**. The array **Cols** has the interpretation that **Col[I] == J** tells us which column the queen in row **I** occupies.

Each **struct** is a task showing one partial solution. The node which obtains this task will then extend this partial solution to several new, longer partial solutions.

The tasks are all placed into a linked list. **Next** points to the next item in the work pool.

A parallel solution based on this idea would like something like this:

```
while (work pool nonempty or at least one nonidle processor) {
   get a TentSoln struct from the work pool, and point P to it;
   I = P->RowsSoFar;
   for (J = 0; J < 8; J++) {
      if (a queen at row I, column J would not attack the previous queens) {
         Q = malloc(sizeof(struct TentSoln));
         Q->RowsSoFar = I+1;
         add the struct pointed to by Q to the work pool;
      }
   }
}
```

There of course would also be code in the case I = 8 to check and see if we have found a solution, and if so, to record it, etc.

Note that any rotation of a solution—interchanging rows and columns—is also a solution. Similarly, any reflection across one of the two main diagonals of the board is also a solution. This information could be used to speed up computation, though at the expense of additionality complexity of the code.

9.3 The 8-Square Puzzle Problem

This game was invented more than 100 years ago. Here is what a typical board position looks like:

0	5	3
1	4	
7	2	6

(The real puzzle has numbering from 1 to 8, but we use 0 to 7.)

Each number is on a little movable square, which can be moved up, down, left and right as long as the spot in the given direction is unoccupied. In the example above, the square 3, for instance, could be moved downward, producing an empty spot at the top right of the puzzle. The object of the game is arrange the squares in ascending numerical order, with square 0 at the upper left of the puzzle (which in this example happens to be the case already).

We again solve this by setting up a work pool, in this case a pool of board positions. Each board position would be implemented in something like this:

```
struct BoardPos   {
   int Row[9];
   int Col[9];
   struct BoardPos *Next;
}
```

Here **Row[I]** and **Col[I]** would be the position of the square numbered **I**. For convenience, we also store the location of the blank position, in **Row[8]** and **Col[8]**.

Suppose a processor goes to the work pool and gets the board position depicted above. In the simplest form of the algorithm, the processor would check each of the three possible moves (4 right, 3 down, 6 up) to see if the resulting board position would duplicate one that had already been checked. All moves that lead to new positions would be added to the work pool. Each processor would loop around, pulling items from the work pool, until some processor somewhere finds a solution to the game (in which case that processor would add termination messages to the work pool, so that the other processors knew to stop). An outline of the algorithm would be as follows:

```
while (work pool nonempty or at least one nonidle processor) {
   get a BoardPos struct from the work pool, and point P to it;
   for (I = 0; I < 8; I++)  {
      for all possible moves of square I do  {
         Q = malloc(sizeof(struct BoardPos));
         fill in *Q according to this move;
         if *Q has not already been checked
            add this board to the work pool;
      }
   }
}
```

Again, code would need to be included for checking to see if a solution has been found, whether we have found that no solution exists, and so on.

Note the operation

```
         if *Q has not already been checked
            add this board to the work pool;
```

Clearly this is needed, to avoid endless cycling. But it is not as inoccuous as it looks. If the set of all previously-checked board positions is to be made available to all processors, this may produce substantial increases in contention for memory and interprocessor interconnects. On the other hand, we could arrange the code such that only certain processors have to know about certain subsets of the set of previously-checked board positions, but this makes the code more complex and may produce load-balancing problems.

A more sophisticated version of the algorithm would use a **branch-and-bound** technique. The idea here is to reduce computation by giving priority in the work pool to those board positions which appear "promising" by some reasonable measure. For example, we could take as our measure the "distance" between a given board position and the goal board position, as defined by the sum of the distances from each numbered square to its place in the winning position. In the example above, for instance, the square numbered 5 is a distance of 2 from its ultimate place (2 meaning, one square to the right, one square down, so 1+1 = 2). The board above is a distance 15 from the winning board.

The idea, then would be that we implement the work pool as an ordered linked list (or other ordered structure), and when a board position is added to the work pool, we insert it according to its distance from the winning board. This way the processors will usually work on the more promising boards, and thus hopefully reach the solution faster.

9.4 Itemset Analysis in Data Mining

9.4.1 What Is It?

The term **data mining** is a buzzword, but all it means is the process of finding relationships among a set of variables. In other words, it would seem to simply be a good old-fashioned statistics problem.

Well, in fact it *is* simply a statistics problem—but writ large. Instead of the tiny sample sizes of 25 you likely saw in your statistics courses, typical sample sizes in the data mining arena run in the hundreds of thousands or even hundreds of millions. And there may be hundreds of variables, in constrast to the, say, half dozen you might see in a statistics course.

Major, Major Warning: With so many variables, the chances of picking up spurious relations between variables is large. And although many books and tutorials on data mining will at least pay lip service to this issue (referring to it as **overfitting**, they don't emphasize it enough.[2]

Putting the overfitting problem aside, though, by now the reader's reaction should be, "This calls for parallel processing," and he/she is correct. Here we'll look at parallelizing a particular problem, called **itemset analysis**, the most famous example of which is the **market basket problem**:

[2]Some writers recommend splitting one's data into a **training set**, which is used to discover relationships, and a **validation set**, which is used to confirm those relationships. However, overfitting can still occur even with this precaution.

9.4.2 The Market Basket Problem

Consider an online bookstore has records of every sale on the store's site. Those sales may be represented as a matrix S, whose (i,j)th element S_{ij} is equal to either 1 or 0, depending on whether the ith sale included book j, i = 0,1,...,s-1, j = 0,1,...,t-1. So each row of S represents one sale, with the 1s in that row showing which titles were bought. Each column of S represents one book title, with the 1s showing which sales transactions included that book.

Let's denote the entire line of book titles by $T_0, ..., T_{b-1}$. An **itemset** is just a subset of this. A **frequent** itemset is one which appears in many of sales transactions. But there is more to it than that. The store wants to choose some books for special ads, of the form "We see you bought books X and Y. We think you may be interested in Z."

Though we are using marketing as a running example here (which is the typical way that this subject is introduced), we will usually just refer to "items" instead of books, and to "database records" rather than sales transactions.

We have the following terminology:

- An **association rule** $I \rightarrow J$ is simply an ordered pair of disjoint itemsets I and J.
- The **support** of an an association rule $I \rightarrow J$ is the proportion of records which include both I and J.
- The **confidence** of an association rule $I \rightarrow J$ is the proportion of records which include J, *among those records which include I.*

Note that in probability terms, the support is basically P(I and J) while the confidence is P(J|I). If the confidenc the book business, it means that buyers of the books in set I also tend to buy those in J. But this information is not very useful if the support is low, because it means that the combination occurs so rarely that it's not worth our time to deal with it.

So, the user—let's call him/her the "data miner"—will first set thresholds for support and confidence, and then set out to find all association rules for which support and confidence exceed their respective thresholds.

9.4.3 Serial Algorithms

Various algorithms have been developed to find frequent itemsets and association rules. The most famous one for the former task is the **Apriori** algorithm. Even it has many forms. We will discuss one of the simplest forms here.

The algorithm is basically a breadth-first tree search. At the root we find the frequent 1-item itemsets. At the second level, we find the frequent 2-item itemsets, and so on. After we finish with level i, we then generate new candidate itemsets of size i+1 from the frequent itemsets we found of size i, by

The key point in the latter operation is that if an itemset is not frequent, i.e. has support less than the threshold, then adding further items to it will make it even less frequent. That itemset is then pruned from the tree, and the branch ends.

Here is the pseudocode:

```
set $F_1$ to the set of 1-item itemsets whose support exceeds the threshold
for i = 2 to b
    $F_i = \phi$
    for each I in $F_{i-1}$
        for each K in $F_1$
            $Q = I \cup K$
            if support(Q) exceeds support threshold
               add Q to $F_i$
    if $F_i$ is empty break
return $\cup_i F_i$
```

Again, there are many refinements of this, which shave off work to be done and thus increase speed. For example, we should avoid checking the same itemsets twice, e.g. first $\{1,2\}$ then $\{2,1\}$. This can be accomplished by keeping itemsets in lexicographical order. We will not pursue any refinements here.

9.4.4 Parallelizing the Apriori Algorithm

Clearly there is lots of opportunity for parallelizing the serial algorithm above. Both of the inner **for** loops can be parallelized in straightforward ways; they are "embarrassingly parallel." There are of course critical sections to worry about in the shared-memory setting, and in the message-passing setting one must designate a manager node in which to store the F_i.

However, as more and more refinements are made in the serial algorithm, then the parallelism in this algorithm become less and less "embarrassing." And things become more challenging if the storage needs of the F_i, and of their associated "accounting materials" such as a directory showing the current tree structure (done via hash trees), become greater than what can be stored in the memory of one node.

In other words, parallelizing the market basket problem can be very challenging. The interested reader is referred to the considerable literature which has developed on this topic.

Chapter 10

Introduction to Parallel Sorting

Sorting is one of the most common operations in parallel processing applications. For example, it is central to many parallel database operations, and important in areas such as image processing, statistical methodology and so on. A number of different types of parallel sorting schemes have been developed. Here we look at some of these schemes.

10.1 Quicksort

You are probably familiar with the idea of quicksort: First break the original array into a "small-element" pile and a "large-element" pile, by comparing to a **pivot** element. In a naive implementation, the first element of the array serves as the pivot, but better performance can be obtained by taking, say, the median of the first three elements. Then "recurse" on each of the two piles, and then string the results back together again.

This is an example of the **divide and conquer** approach seen in so many serial algorithms. It is easily parallelized (though load-balancing issues may arise). Here, for instance, we might assign one pile to one thread and the other pile to another thread.

Suppose the array to be sorted is named **x**, and consists of **n** elements.

In a naive implementation, the piles would be put into new arrays, but this is bad in two senses: It wastes memory space, and wastes time, since much copying of arrays needs to be done. A better implementation places the two piles back into the original array **x**. The following C code does that.

The function **separate()** is intended to be used in a recursive quicksort operation. It operates on **x[l]** through **x[h]**, a subarray of **x** that itself may have been formed at an earlier stage of the recursion. It forms two piles from those elements, and placing the piles back in the same region **x[l]** through **x[h]**. It also has a return value, showing where the first pile ends.

```
int separate(int l, int h)
{  int ref,i,j,k,tmp;
   ref = x[h]; i = l-1; j = h;
   do  {
      do i++; while (x[i] < ref && i < h);
      do j--; while (x[j] > ref && j > l);
      tmp = x[i];  x[i] = x[j];  x[j] = tmp;
   } while (j > i);
   x[j] = x[i];  x[i] = x[h];  x[h] = tmp;
   return i;
}
```

The function **separate()** rearranges the subarray, returning a value **m**, so that:

- **x[l]** through **x[m-1]** are less than **x[m]**,
- **x[m+1]** through **x[h]** are greater than **x[m]**, and
- **x[m]** is in its "final resting place," meaning that **x[m]** will never move again for the remainder of the sorting process. (Another way of saying this is that the current **x[m]** is the **m**-th smallest of all the original **x[i]**, **i** = 0,1,...,**n**-1.)

By the way, **x[l]** through **x[m-1]** will also be in their final resting places as a group. They may be exchanging places with each other from now on, but they will never again leave the range **i** though **m-1** within the **x** array as a whole. A similar statement holds for **x[m+1]** through **x[n-1]**.

10.1.1 Shared-Memory Quicksort

Here is OpenMP code which performs quicksort in the shared-memory paradigm (adapted from code in the OpenMP Source Code Repository, `http://www.pcg.ull.es/ompscr/`):

```
void qs(int *x, int l, int h)
{  int newl[2], newh[2], i, m;
   m = separate(x,l,h);
   newl[0] = l;  newh[0] = m-1;
   newl[1] = m+1;  newh[1] = h;
   #pragma omp parallel
   {
      #pragma omp for nowait
      for (i = 0; i < 2; i++)
         qs(newl[i],newh[i]);
   }
}
```

Note the **nowait** clause. Since different threads are operating on different portions of the array, they need not be synchronized.

A variant on this which might achieve better load balancing would set up a **task pool**, consisting of an array of (**l**, **h**) pairs. Initially the pool consists of just [0,n-1]. The function **qs()** would then become iterative instead of recursive, with its main loop looking something like this for an array of length n:

```
fetch an (l,h) pair from the task pool
while not done
   call separate() on x[l] through x[h], yielding m
   if m < h
      add (m+1,h) to the task pool
   h = m-1
   if l == h
      fetch [l,h] from the task pool
```

This pseudocode is missing important details. For example, How does the iteration within a thread stop? The key lies in pairs of the form (i,i), which I'll call *singletons*. The sort is done when the number of singletons reaches n.

10.1.2 Hyperquicksort

This algorithm was originally developed for hypercubes, but can be used on any message-passing system having a power of 2 for the number of nodes.[1]

It is assumed that at the beginning each PE contains some chunk of the array to be sorted. After sorting, each PE will contain some chunk of the sorted array, meaning that:

- each chunk is itself in sorted form
- for all cases of $i < j$, the elements at PE i are less than the elements at PE j

If the sorted array itself were our end, rather than our means to something else, we could now collect it at some node, say node 0. If, as is more likely, the sorting is merely an intermediate step in a larger distributed computation, we may just leave the chunks at the nodes and go to the next phase of work.

Say we are on a d-cube. The intuition behind the algorithm is quite simple:

```
for i = d downto 1
   for each i-cube:
      root of the i-cube broadcasts its median to all in the i-cube,
         to serve as pivot
      consider the two (i-1)-subcubes of this i-cube
      each pair of partners in the (i-1)-subcubes exchanges data:
         low-numbered PE gives its partner its data larger than pivot
         high-numbered PE gives its partner its data smaller than pivot
```

[1]See Chapter 6 for definitions of hypercube terms.

To avoid deadlock, have the lower-numbered partner send then receive, and vice versa for the higher-numbered one. Better, in MPI, use **MPI_SendRcv()**.

After the first iteration, all elements in the lower (d-1)-cube are less than all elements in higher (d-1)-cube. After d such steps, the array will be sorted.

10.2 Mergesorts

10.2.1 Sequential Form

In its serial form, mergesort has the following pseudocode:

```
// initially called with l = 0 and h = n-1, where n is the length of the
// array and is assumed here to be a power of 2
void seqmergesort(int *x, int l, int h)
{  seqmergesort(x,0,h/2-1);
   seqmergesort(x,h/2,h);
   merge(x,l,h);
}
```

The function **merge()** should be done in-place, i.e. without using an auxiliary array. It basically codes the operation shown in pseudocode for the message-passing case in Section 10.2.3.

10.2.2 Shared-Memory Mergesort

This is similar to the patterns for shared-memory quicksort in Section 10.1.1 above.

10.2.3 Message Passing Mergesort on a Tree Topology

First, we organize the processing nodes into a binary tree. This is simply from the point of view of the software, rather than a physical grouping of the nodes. We will assume, though, that the number of nodes is one less than a power of 2.

To illustrate the plan, say we have seven nodes in all. We could label node 0 as the root of the tree, label nodes 1 and 2 to be its two children, label nodes 3 and 4 to be node 1's children, and finally label nodes 5 and 6 to be node 2's children.

It is assumed that the array to be sorted is initially distributed in the leaf nodes (recall a similar situation for hyperquicksort), i.e. nodes 3-6 in the above example. The algorithm works best if there are approximately the same number of array elements in the various leaves.

In the first stage of the algorithm, each leaf node applies a regular sequential sort to its current holdings. Then each node begins sending its now-sorted array elements to its parent, one at a time, in ascending numerical order.

Each nonleaf node then will merge the lists handed to it by its two children. Eventually the root node will have the entire sorted array. Specifically, each nonleaf node does the following:

```
do
   if my left-child datum < my right-child datum
      pass my left-child datum to my parent
   else
      pass my right-child datum to my parent
until receive the "no more data" signal from both children
```

Of course, due to network latency and the like, one may get better performance if each node accumulates a chunk of data before sending to the parent, rather than sending just one datum at a time.

10.2.4 Compare-Exchange Operations

These are key to many sorting algorithms.

A **compare-exchange**, also known as **compare-split**, simply means in English, “Let’s pool our data, and then I’ll take the lower half and you take the upper half.” Each node executes the following pseudocode:

```
send all my data to partner
receive all my partner’s data
if I have a lower id than my partner
   I keep the lower half of the pooled data
else
   I keep the upper half of the pooled data
```

10.2.5 Bitonic Mergesort

A sequence $(a_0, a_1, .., a_{k-1})$ is called **bitonic** if it is first nondecreasing then nonincreasing, meaning that for some r

$$(a_0 \leq a_1 \leq ... \leq a_r \geq a_{r+1} \geq a_{n-1})$$

(For convenience, from here on I will use the terms *increasing* and *decreasing* instead of *nonincreasing* and *nondecreasing*.)

This includes the cases in which the sequence is purely nondecreasing (r = n-1) or purely nonincreasing (r = 0) . By convention, it also includes sequences which can be cyclically shifted into the above form.

For instance, the sequence (3,8,12,15,14,5,1,2) can be rotated rightward by two element positions to form (1,2,3,8,12,15,14,5), so (3,8,12,15,14,5,1,2) is defined to be bitonic too.

These are the "A-type" bitonic sequences, so called because they look like the letter A (or like a carat). The "V-type" bitonic sequences consist of a nonincreasing sequence followed by a nondecreasing sequence.

Suppose we have bitonic sequence $(a_0, a_1, .., a_{k-1})$, where k is a power of 2. Rearrange the sequence by doing compare-exchange operations between a_i and $a_{n/2+i}$), i = 0,1,...,n/2-1. Then it is not hard to prove that the new $(a_0, a_1, .., a_{k/2-1})$ and $(a_{k/2}, a_{k/2+1}, .., a_{k-1})$ are bitonic, and every element of that first subarray is less than or equal to every element in the second one.

So, we have set things up for yet another divide-and-conquer attack:

```
// x is bitonic of length n, n a power of 2
void sortbitonic(int *x, int n)
{  do the pairwise compare-exchange operations
   if (n > 2) {
      sortbitonic(x,n/2);
      sortbitonic(x+n/2,n/2);
   }
}
```

So much for sorting bitonic sequences. But what about general sequences? We can proceed as follows:

1. Each of the pairs (a_i, a_{i+1}), i = 0,2,...,n-2 is bitonic, since *any* 2-element array is bitonic!

2. For each i = 0,2,4,...,n-2:

    - Apply **sortbitonic()** to (a_i, a_{i+1}).
    - If i/2 is odd, reverse the pair, so that this pair and the pair immediately preceding it now form a 4-element bitonic sequence.

3. For each i = 0,4,8,...,n-4:

    - Apply **sortbitonic()** to $(a_i, a_{i+1}, a_{i+2}, a_{i+3})$.
    - If i/4 is odd, reverse the quartet, so that this quartet and the quartet immediately preceding it now form an 8-element bitonic sequence.

4. Keep building in this manner, until get to a single sorted n-element list.

There are many ways to parallelize this. In the hypercube case, the algorithm consists of doing compare-exchange operations with all neighbors, pretty much in the same pattern as hyperquicksort.

10.3 The Bubble Sort and Its Cousins

10.3.1 The Much-Maligned Bubble Sort

Recall the **bubble sort**:

```
void bubblesort(int *x, int n)
{  for i = n-1 downto 1
      for j = 0 to i
         compare-exchange(x,i,j,n)
}
```

Here the function **compare-exchange()** is as in Section 10.2.4 above. In the context here, it boils down to

```
if x[i] > x[j]
   swap x[i] and x[j]
```

In the first **i** iteration, the largest element "bubbles" all the way to the right end of the array. In the second iteration, the second-largest element bubbles to the next-to-right-end position, and so on.

You learned in your algorithms class that this is a very inefficient algorithm—when used serially. But it's actually rather usable in parallel systems.

For example, in the shared-memory setting, suppose we have one thread for each value of **i**. Then those threads can work in parallel, as long as a thread with a larger value of **i** does not overtake a thread with a smaller **i**, where "overtake" means working on a larger **j** value.

Once again, it probably pays to chunk the data. In this case, **compare-exchange()** fully takes on the meaning it had in Section 10.2.4.

10.3.2 A Popular Variant: Odd-Even Transposition

A popular variant of this is the **odd-even transposition sort**. The pseudocode for a shared-memory version is:

```
// the argument "me" is this thread's ID
void oddevensort(int *x, int n, int me)
{  for i = 1 to n
      if i is odd
         if me is even
           compare-exchange(x,me,me+1,n)
         else  // me is odd
           compare-exchange(x,me,me-1,n)
      else  // i is even
```

```
        if me is even
          compare-exchange(x,me,me-1,n)
        else  // me is odd
          compare-exchange(x,me,me+1,n)
```

If the second or third argument of **compare-exchange()** is less than 0 or greater than **n**-1, the function has no action.

This looks a bit complicated, but all it's saying is that, from the point of view of an even-numbered element of **x**, it trades with its right neighbor during odd phases of the procedure and with its left neighbor during even phases.

Again, this is usually much more effective if done in chunks.

10.4 Shearsort

In some contexts, our hardware consists of a two-dimensional mesh of PEs. A number of methods have been developed for such settings, one of the most well known being Shearsort, developed by Sen, Shamir and the eponymous Isaac Scherson of UC Irvine. Again, the data is assumed to be initially distributed among the PEs. Here is the pseudocode:

```
for i = 1 to ceiling(log2(n)) + 1
    if i is odd
       sort each even row in descending order
       sort each odd row in ascending order
    else
       sort each column is ascending order
```

At the end, the numbers are sorted in a "snakelike" manner.

For example:

6	12
5	9

6	12
9	5

6	5
9	12

5	6 ↓
12	← 9

No matter what kind of system we have, a natural domain decomposition for this problem would be for each process to be responsible for a group of rows. There then is the question about what to do during the even-numbered iterations, in which column operations are done. This can be handled via a parallel matrix transpose operation. In MPI, the function **MPI_Alltoall()** may be useful.

10.5 Bucket Sort with Sampling

For concreteness, suppose we are using MPI on message-passing hardware, say with 10 PEs. As usual in such a setting, suppose our data is initially distributed among the PEs.

Suppose we knew that our array to be sorted is a random sample from the uniform distribution on (0,1). In other words, about 20% of our array will be in (0,0.2), 38% will be in (0.45,0.83) and so on.

What we could do is assign PE0 to the interval (0,0.1), PE1 to (0.1,0.2) etc. Each PE would look at its local data, and distribute it to the other PEs according to this interval scheme. Then each PE would do a local sort.

In general, we don't know what distribution our data comes from. We solve this problem by doing sampling. In our example here, each PE would sample some of its local data, and send the sample to PE0. From all of these samples, PE0 would find the decile values, i.e. 10th percentile, 20th percentile,..., 90th percentile. These values, called **splitters** would then be broadcast to all the PEs, and they would then distribute their local data to the other PEs according to these intervals.

Chapter 11

Parallel Computation of Fourier Series, with an Introduction to Parallel Imaging

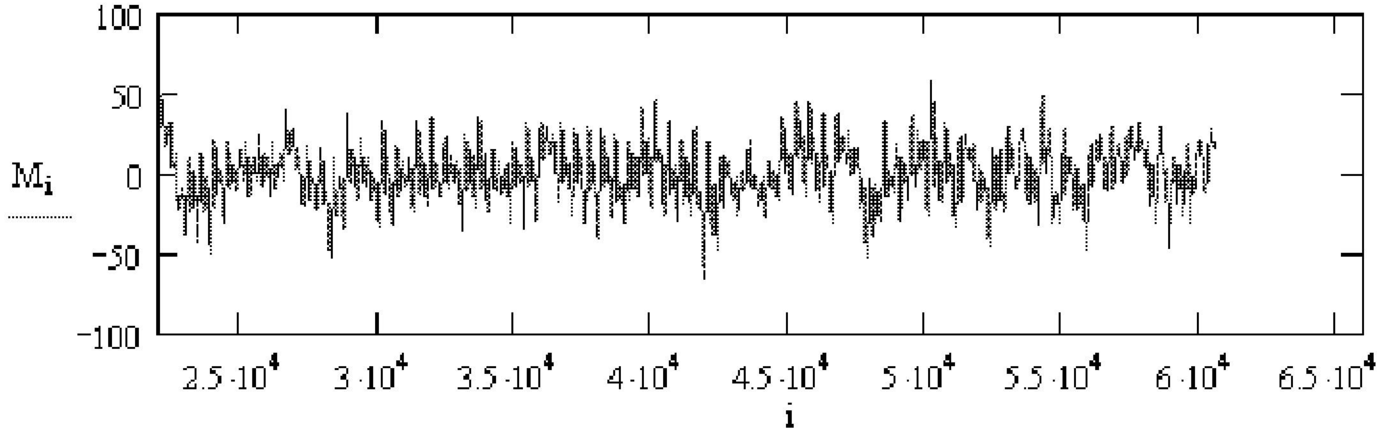
Mathematical computations involving sounds and images, for example for voice and pattern recognition are often performed using **Fourier** analysis.

11.1 General Principles

11.1.1 One-Dimensional Fourier Series

A sound **wave form** graphs volume of the sound against time. Here, for instance, is the wave form for a vibrating reed:[1]

[1]Reproduced here by permission of Prof. Peter Hamburger, Indiana-Purdue University, Fort Wayne. See http://www.ipfw.edu/math/Workshop/PBC.html



Recall that we say a function of time g(t) is **periodic** (“repeating,” in our casual wording above) with period T if if g(u+T) = g(u) for all u. The **fundamental frequency** of g() is then defined to be the number of periods per unit time,

$$f_0 = \frac{1}{T} \tag{11.1}$$

Recall also from calculus that we can write a function g(t) (not necessarily periodic) as a Taylor series, which is an “infinite polynomial”:

$$g(t) = \sum_{n=0}^{\infty} c_n t^n. \tag{11.2}$$

The specific values of the c_n may be derived by differentiating both sides of (11.2) and evaluating at t = 0, yielding

$$c_n = \frac{g^{(n)}(0)}{n!}, \tag{11.3}$$

where $g^{(j)}$ denotes the ith derivative of g().

For instance, for e^t,

$$e^t = \sum_{n=0}^{\infty} \frac{1}{n!} t^n \tag{11.4}$$

In the case of a repeating function, it is more convenient to use another kind of series representation, an “infinite trig polynomial,” called a **Fourier series**. This is just a fancy name for a weighted sum of sines and cosines of different frequencies. More precisely, we can write any repeating function g(t) with period T and fundamental frequency f_0 as

$$g(t) = \sum_{n=0}^{\infty} a_n \cos(2\pi n f_0 t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n f_0 t) \tag{11.5}$$

for some set of weights a_n and b_n. Here, instead of having a weighted sum of terms

$$1,\ t,\ t^2,\ t^3,\ ... \tag{11.6}$$

as in a Taylor series, we have a weighted sum of terms

$$1,\ \cos(2\pi f_0 t),\ \cos(4\pi f_0 t),\ \cos(6\pi f_0 t),\ ... \tag{11.7}$$

and of similar sine terms. Note that the frequencies nf_0, in those sines and cosines are integer multiples of the fundamental frequency of x, f_0, called **harmonics**.

The weights a_n and b_n, n = 0, 1, 2, ... are called the **frequency spectrum** of g(). The coefficients are calculated as follows:[2]

$$a_0 = \frac{1}{T} \int_0^T g(t)\ dt \tag{11.8}$$

$$a_n = \frac{2}{T} \int_0^T g(t)\ cos(2\pi n f_0 t)\ dt \tag{11.9}$$

$$b_n = \frac{2}{T} \int_0^T g(t)\ sin(2\pi n f_0 t)\ dt \tag{11.10}$$

By analyzing these weights, we can do things like machine-based voice recognition (distinguishing one person's voice from another) and speech recognition (determining what a person is saying). If for example one person's voice is higher-pitched than that of another, the first person's weights will be concentrated more on the higher-frequency sines and cosines than will the weights of the second.

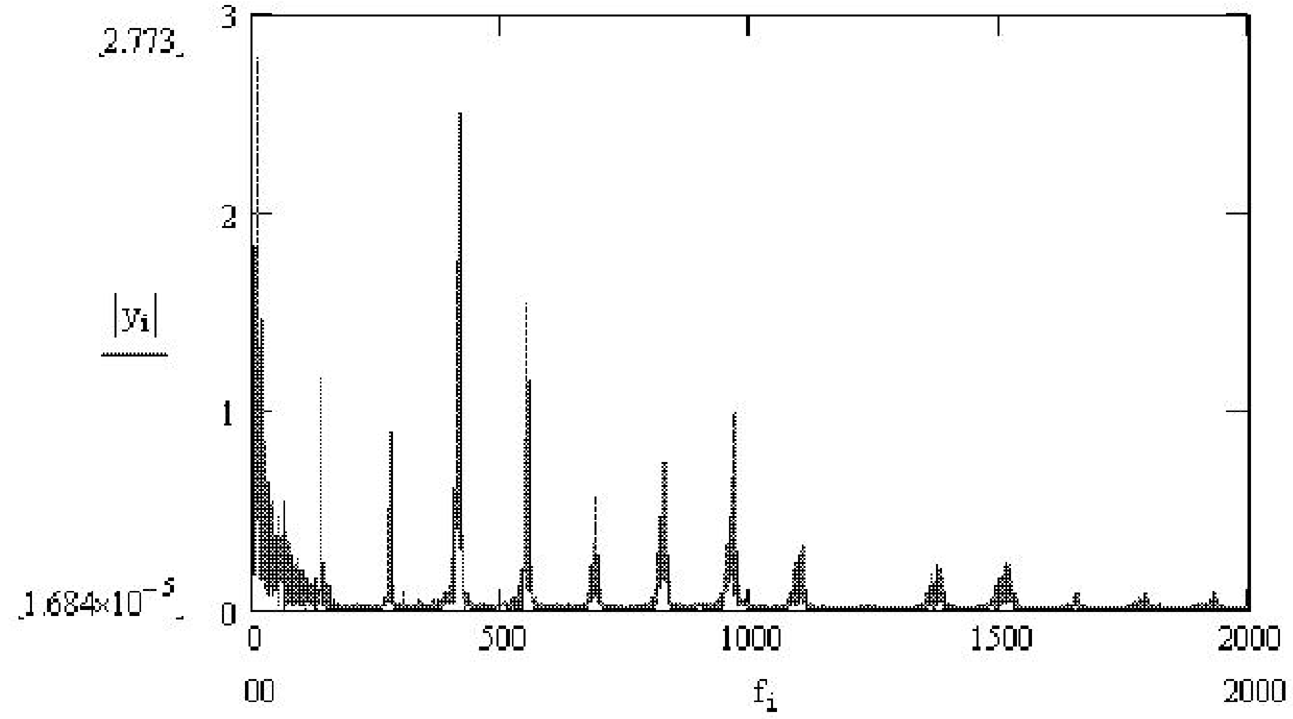
Since g(t) is a graph of loudness against time, this representation of the sound is called the **time domain**. When we find the Fourier series of the sound, the set of weights a_n and b_n is said to be a representation of

[2]The get an idea as to how these formulas arise, see Section 11.8. But for now, if you integrate both sides of (11.5), you will at least verify that the formulas below do work.

the sound in the **frequency domain**. One can recover the original time-domain representation from that of the frequency domain, and vice versa, as seen in Equations (11.8), (11.9), (11.10) and (11.5).

In other words, the transformations between the two domains are inverses of each other, and there is a one-to-one correspondence between them. Every g() corresponds to a unique set of weights and vice versa.

Now here is the frequency-domain version of the reed sound:



Note that this graph is very "spiky." In other words, even though the reed's waveform includes all frequencies, most of the power of the signal is at a few frequencies which arise from the physical properties of the reed.

Fourier series are often expressed in terms of complex numbers, making use of the relation

$$e^{i\theta} = \cos(\theta) + i\ \sin(\theta), \tag{11.11}$$

where $i = \sqrt{-1}$.[3]

[3]There is basically no physical interpretation of complex numbers. Instead, they are just mathematical abstractions. However, they are highly useful abstractions, with the complex form of Fourier series, beginning with (11.12), being a case in point.

The complex form of (11.5) is

$$g(t) = \sum_{j=-\infty}^{\infty} c_j e^{2\pi i j \frac{t}{T}}. \tag{11.12}$$

The c_j are now generally complex numbers. They are functions of the a_j and b_j, and thus comprise the frequency spectrum.

Equation (11.12) has a simpler, more compact form than (11.5). Do you now see why I referred to Fourier series as trig polynomials? The series (11.12) involves the jth powers of $e^{2\pi \frac{t}{T}}$.

11.1.2 Two-Dimensional Fourier Series

Let's now move from sounds images. Here g() is a function of two variables, g(u,v), where u and v are the horizontal and vertical coordinates of a pixel in the image; g(u,v) is the intensity of the image at that pixel. If it is a gray-scale image, the intensity is whiteness of the image at that pixel, typically with 0 being pure black and 255 being pure white. If it is a color image, a typical graphics format is to store three intensity values at a point, one for each of red, green and blue. The various colors come from combining three colors at various intensities.

Since images are two-dimensional instead of one-dimensional like a sound wave form, the Fourier series for an image is a sum of sines and cosines in two variables, i.e. a double sum $\Sigma_j \Sigma_k ...$ instead of $\Sigma_j ...$.

The terminology changes a bit. Our original data is now referred to as being in the **spatial domain**, rather than the time domain. But the Fourier series coefficients are still said to be in the frequency domain.

11.2 Discrete Fourier Transforms

In sound and image applications, we seldom if ever know the exact form of the repeating function g(). All we have is a **sampling** from g(), i.e. we only have values of g(t) for a set of discrete values of t.

In the sound example above, a typical sampling rate is 8000 samples per second.[4] So, we may have g(0), g(0.000125), g(0.000250), g(0.000375), and so on. In the image case, we sample the image pixel by pixel.

Thus we can't calculate integrals like (11.8). So, how do we approximate the Fourier transform based on the sample data?

[4]See Section 11.9 for the reasons behind this.

11.2.1 One-Dimensional Data

Let $X = (x_0, ..., x_{n-1})$ denote the sampled values, i.e. the time-domain representation of g() based on our sample data. These are interpreted as data from one period of g(), with the period being n and the fundamental frequency being 1/n. The frequency-domain representation will also consist of n numbers, $c_0, ..., c_{n-1}$, defined as follows:[5]

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} = \frac{1}{n} \sum_{j=0}^{n-1} x_j q^{jk} \tag{11.13}$$

where

$$q = e^{-2\pi i/n} \tag{11.14}$$

again with $i = \sqrt{-1}$. The array C of complex numbers c_k is called the **discrete Fourier transform** (DFT) of X.

Note that (11.13) is basically a discrete analog of (11.9) and (11.10).

As in the continuous case, we can recover each domain from the other. So, while (11.13) shows how to go to the frequency domain from the time domain, we can go from the frequency domain to the time domain via the inverse transform, whose equation is

$$x_k = \sum_{j=0}^{n-1} c_j e^{2\pi ijk/n} = \sum_{j=0}^{n-1} c_j q^{-jk} \tag{11.15}$$

Note that (11.15) is basically a discrete analog of (11.5).

Note too that instead of having infinitely many harmonics, we can only have n of them: 1, 1/n, 2/n, ..., (n-1)/n. It would be impossible to have more than n, as can be seen by reasoning as follows: The x_k are given, q is a constant, and we are solving for the c_j. So, we have n equations in n unknowns. If we had more than n unknowns, the system would be indeterminate.

[5]It should be noted that there are many variant definitions of these transforms. One common variation is to include/exclude a scale factor, such as our 1/n in (11.13). Another type of variations involve changing only c_0, in order to make certain matrices have more convenient forms.

11.2.2 Two-Dimensional Data

The spectrum numbers c_{rs} are double-subscripted, like the original data x_{uv}, the latter being the pixel intensity in row u, column v of the image, u = 0,1,...,n-1, v = 0,1,...,m-1. Equation (11.13) becomes

$$c_{rs} = \frac{1}{n}\frac{1}{m}\sum_{j=0}^{n-1}\sum_{k=0}^{m-1} x_{jk} e^{-2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{11.16}$$

Its inverse is

$$x_{rs} = \sum_{j=0}^{n-1}\sum_{k=0}^{m-1} c_{jk} e^{2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{11.17}$$

11.3 Parallel Computation of Discrete Fourier Transforms

11.3.1 The Fast Fourier Transform

Speedy computation of a discrete Fourier transform was developed by Cooley and Tukey in their famous Fast Fourier Transform (FFT), which takes a "divide and conquer" approach:

Equation (11.13) can be rewritten as

$$c_k = \frac{1}{n}\left[\sum_{j=0}^{m-1} x_{2j} q^{2jk} + \sum_{j=0}^{m-1} x_{2j+1} q^{(2j+1)k},\right] \tag{11.18}$$

where $m = n/2$.

After some algebraic manipulation, this becomes

$$c_k = \frac{1}{2}\left[\frac{1}{m}\sum_{j=0}^{m-1} x_{2j} z^{jk} + q^k \frac{1}{m}\sum_{j=0}^{m-1} x_{2j+1} z^{jk}\right] \tag{11.19}$$

where $z = e^{-2\pi i/m}$.

A look at Equation (11.19) shows that the two sums within the brackets have the same form as Equation (11.13). In other words, Equation (11.19) shows how we can compute an n-point FFT from two $\frac{n}{2}$-point

FFTs. That means that a DFT can be computed recursively, cutting the sample size in half at each recursive step.

In a shared-memory setting such as OpenMP, we could implement this recursive algorithm in the manners of Quicksort in Chapter 10.

In a message-passing setting, one can use the butterfly algorithm, explained for implementation of barriers in Chapter 1. Some digital signal processing chips implement this in hardware, with a special interconnection network to implement this algorithm.

11.3.2 A Matrix Approach

The matrix form of (11.13) is

$$C = \frac{1}{n} AX \tag{11.20}$$

where A is n x n. Element (j,k) of A is q^{jk}, while element j of X is x_j. This formulation of the problem then naturally leads one to use parallel methods for matrix multiplication; see Chapter 8.

11.3.3 Parallelizing Computation of the Inverse Transform

The form of the DFT (11.13) and its inverse (11.15) are very similar. For example, the inverse transform is again of a matrix form as in (11.20); even the new matrix looks a lot like the old one.[6]

Thus the methods mentioned above, e.g. FFT and the matrix approach, apply to calculation of the inverse transforms too.

11.3.4 Parallelizing Computation of the Two-Dimensional Transform

Regroup (11.16) as:

$$c_{rs} = \frac{1}{n} \sum_{j=0}^{n-1} \left(\frac{1}{m} \sum_{k=0}^{m-1} x_{jk} e^{-2\pi i (\frac{ks}{m})} \right) e^{-2\pi i (\frac{jr}{n})} \tag{11.21}$$

$$= \frac{1}{n} \sum_{j=0}^{n-1} y_{js} e^{-2\pi i (\frac{jr}{n})} \tag{11.22}$$

[6]In fact, one can obtain the new matrix easily from the old, as explained in Section 11.8.

Note that y_{js}, i.e. the expression between the large parentheses, is the sth component of the DFT of the jth row of our data. And hey, the last expression (11.22) above is in the same form as (11.13)! Of course, this means we are taking the DFT of the spectral coefficients rather than observed data, but numbers are numbers.

In other words: To get the two-dimensional DFT of our data, we first get the one-dimensional DFTs of each row of the data, place these in rows, and then find the DFTs of each column. This property is called **separability**.

This certainly opens possibilities for parallelization. Each thread (shared memory case) or node (message passing case) could handle groups of rows of the original data, and in the second stage each thread could handle columns.

Or, we could interchange rows and columns in this process, i.e. put the j sum inside and k sum outside in the above derivation.

11.4 Applications to Image Processing

In image processing, there are a number of different operations which we wish to perform. We will consider two of them here.

11.4.1 Smoothing

An image may be too "rough." There may be some pixels which are noise, accidental values that don't fit smoothly with the neighboring points in the image.

One way to smooth things out would be to replace each pixel intensity value[7] by the mean or median among the pixels neighbors. These could be the four immediate neighbors if just a little smoothing is needed, or we could go further out for a higher amount of smoothing. There are many variants of this.

But another way would be to apply a **low-pass filter** to the DFT of our image. This means that after we compute the DFT, we simply delete the higher harmonics, i.e. set c_{rs} to 0 for the larger values of r and s. We then take the inverse transform back to the spatial domain. Remember, the sine and cosine functions of higher harmonics are "wigglier," so you can see that all this will have the effect of removing some of the wiggliness in our image—exactly what we wanted.

We can control the amount of smoothing by the number of harmonics we remove.

The term *low-pass filter* obviously alludes to the fact that the low frequencies "pass" through the filter but the high frequencies are blocked. Since we've removed the high-oscillatory components, the effect is a

[7]Remember, there may be three intensity values per pixel, for red, green and blue.

smoother image.[8]

To do smoothing in parallel, if we just average neighbors, this is easily parallelized. If we try a low-pass filter, then we use the parallelization methods shown here earlier.

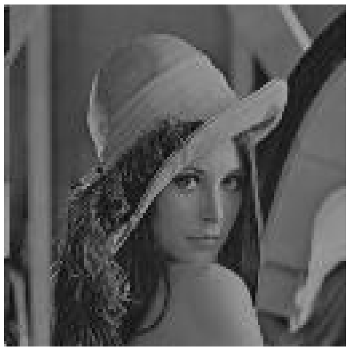
11.4.2 Edge Detection

In computer vision applications, we need to have a machine-automated way to deduce which pixels in an image form an edge of an object.

Again, edge-detection can be done in primitive ways. Since an edge is a place in the image in which there is a sharp change in the intensities at the pixels, we can calculate slopes of the intensities, in the horizontal and vertical directions. (This is really calculating the approximate values of the partial derivatives in those directions.)

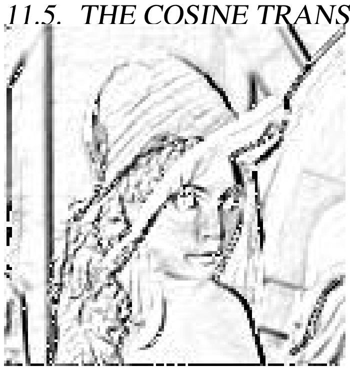
But the Fourier approach would be to apply a high-pass filter. Since an edge is a set of pixels which are abruptly different from their neighbors, we want to keep the high-frequency components and block out the low ones.

Below we have "before and after" pictures, first of original data and then the picture after an edge-detection process has been applied.[9]



[8]Note that we may do more smoothing in some parts of the image than in others.

[9]These pictures are courtesy of Bill Green of the Robotics Laboratory at Drexel University. In this case he is using a Sobel process instead of Fourier analysis, but the result would have been similar for the latter. See his Web tutorial at `www.pages.drexel.edu/~weg22/edge.html`.



The second picture looks like a charcoal sketch! But it was derived mathematically from the original picture, using edge-detection methods.

Note that edge detection methods also may be used to determine where sounds ("ah," "ee") begin and end in speech-recognition applications. In the image case, edge detection is useful for face recognition, etc.

Parallelization here is similar to that of the smoothing case.

11.5 The Cosine Transform

It's inconvenient, to say the least, to work with all those complex numbers. But an alternative exists in the form of the **cosine transform**, which is a linear combination of cosines in the one-dimensional case, and of products of cosines in the two-dimensional case.

$$d_{uv} = \frac{2}{\sqrt{mn}} Y(u)Y(v) \sum_{j=0}^{n-1} \sum_{k=0}^{m-1} x_{jk} \, cos \frac{(2j+1)u\pi}{2n} cos \frac{(2k+1)v\pi}{2m}, \tag{11.23}$$

where $Y(0) = 1/\sqrt{2}$ and $Y(t) = 1$ for $t > 0$.

$$x_{jk} = \frac{2}{\sqrt{mn}} \sum_{u=0}^{n-1} \sum_{v=0}^{m-1} Y(u)Y(v) d_{uv} \, cos \frac{(2j+1)u\pi}{2n} cos \frac{(2k+1)v\pi}{2m}, \tag{11.24}$$

11.6 Keeping the Pixel Intensities in the Proper Range

Normally pixel intensities are stored as integers between 0 and 255, inclusive. With many of the operations mentioned above, both Fourier-based and otherwise, we can get negative intensity values, or values higher than 255. We may wish to discard the negative values and scale down the positive ones so that most or all are smaller than 256.

Furthermore, even if most or all of our values are in the range 0 to 255, they may be near 0, i.e. too faint. If so, we may wish to multiply them by a constant.

11.7 Does the Function g() Really Have to Be Repeating?

It is clear that in the case of a vibrating reed, our loudness function g(t) really is periodic. What about other cases?

A graph of your voice would look "locally periodic." One difference would be that the graph would exhibit more change through time as you make various sounds in speaking, compared to the one repeating sound for the reed. Even in this case, though, your voice *is* repeating within short time intervals, each interval corresponding to a different sound. If you say the word *eye*, for instance, you make an "ah" sound and then an "ee" sound. The graph of your voice would show one repeating pattern during the time you are saying "ah," and another repeating pattern during the time you are saying "ee." So, even for voices, we do have repeating patterns over short time intervals.

On the other hand, in the image case, the function may be nearly constant for long distances (horizontally or vertically), so a local periodicity argument doesn't seem to work there.

The fact is, though, that it really doesn't matter in the applications we are considering here. Even though mathematically our work here has tacitly assumed that our image is duplicated infinitely times (horizontally and vertically),[10] we don't care about this. We just want to get a measure of "wiggliness," and fitting linear combinations of trig functions does this for us.

11.8 Vector Space Issues (optional section)

The theory of Fourier series (and of other similar transforms), relies on vector spaces. It actually is helpful to look at some of that here. Let's first discuss the derivation of (11.13).

Define X and C as in Section 11.2. X's components are real, but it is also a member of the vector space V of all n-component arrays of complex numbers.

For any complex number a+bi, define its **conjugate**, $\overline{a+bi} = a - bi$. Note that

$$\overline{e^{i\theta}} = \cos\theta - i\sin\theta == \cos(-\theta) + i\sin(-\theta) = e^{-i\theta} \tag{11.25}$$

[10]And in the case of the cosine transform, implicitly we are assuming that the image flips itself on every adjacent copy of the image, first right-side up, then upside-own, then right-side up again, etc.

Define an inner product ("dot product"),

$$[u, w] = \frac{1}{n} \sum_{j=0}^{n-1} u_j \bar{w}_j. \tag{11.26}$$

Define

$$v_h = (1, q^{-h}, q^{-2h}, ..., q^{-(n-1)h}), h = 0, 1, ..., n-1. \tag{11.27}$$

Then it turns out that the v_h form an orthonormal basis for V.[11] For example, to show orthnogonality, observe that for $r \neq s$

$$\begin{aligned}
[v_r, v_s] &= \frac{1}{n} \sum_{j=0}^{n-1} v_{rj} \overline{v_{sj}} && (11.28)\\
&= \frac{1}{n} \sum_{j=0} q^{j(-r+s)} && (11.29)\\
&= \frac{1 - q^{(-r+s)n}}{n(1-q)} && (11.30)\\
&= 0, && (11.31)
\end{aligned}$$

due to the identity $1 + y + y^2 + ... + y^k = \frac{1-y^{k+1}}{1-y}$ and the fact that $q^n = 1$. In the case r = s, the above computation shows that $[v_r, v_s] = 1$.

The DFT of X, which we called C, can be considered the "coordinates" of X in V, relative to this orthonormal basis. The kth coordinate is then $[X, v_k]$, which by definition is (11.13).

The fact that we have an orthonormal basis for V here means that the matrix A/n in (11.20) is an orthogonal matrix. For real numbers, this means that this matrix's inverse is its transpose. In the complex case, instead of a straight transpose, we do a conjugate transpose, $B = \overline{A/n}^t$, where t means transpose. So, B is the inverse of A/n. In other words, in (11.20), we can easily get back to X from C, via

$$X = BC = \frac{1}{n} \bar{A}^t C. \tag{11.32}$$

It's really the same for the nondiscrete case. Here the vector space consists of all the possible periodic functions g() (with reasonable conditions placed regarding continuity etc.) forms the vector space, and the

[11]Recall that this means that these vectors are orthogonal to each other, and have length 1, and that they span V.

sine and cosine functions form an orthonormal basis. The a_n and b_n are then the “coordinates” of g() when the latter is viewed as an element of that space.

11.9 Bandwidth: How to Read the *San Francisco Chronicle* Business Page (optional section)

The popular press, especially business or technical sections, often uses the term **bandwidth**. What does this mean?

Any transmission medium has a natural range [f_{min},f_{max}] of frequencies that it can handle well. For example, an ordinary voice-grade telephone line can do a good job of transmitting signals of frequencies in the range 0 Hz to 4000 Hz, where “Hz” means cycles per second. Signals of frequencies outside this range suffer fade in strength, i.e are **attenuated**, as they pass through the phone line.[12]

We call the frequency interval [0,4000] the **effective bandwidth** (or just the **bandwidth**) of the phone line.

In addition to the bandwidth of a **medium**, we also speak of the bandwidth of a **signal**. For instance, although your voice is a mixture of many different frequencies, represented in the Fourier series for your voice’s waveform, the really low and really high frequency components, outside the range [340,3400], have very low power, i.e. their a_n and b_n coefficients are small. Most of the power of your voice signal is in that range of frequencies, which we would call the effective bandwidth of your voice waveform. This is also the reason why digitized speech is sampled at the rate of 8,000 samples per second. A famous theorem, due to Nyquist, shows that the sampling rate should be double the maximum frequency. Here the number 3,400 is “rounded up” to 4,000, and after doubling we get 8,000.

Obviously, in order for your voice to be heard well on the other end of your phone connection, the bandwidth of the phone line must be at least as broad as that of your voice signal, and that is the case.

However, the phone line’s bandwidth is not much broader than that of your voice signal. So, some of the frequencies in your voice will fade out before they reach the other person, and thus some degree of distortion will occur. It is common, for example, for the letter ‘f’ spoken on one end to be mis-heard as ‘s’on the other end. This also explains why your voice sounds a little different on the phone than in person. Still, most frequencies are reproduced well and phone conversations work well.

We often use the term “bandwidth” to literally refer to width, i.e. the width of the interval $[f_{min}, f_{max}]$.

There is huge variation in bandwidth among transmission media. As we have seen, phone lines have bandwidth intervals covering values on the order of 10^3. For optical fibers, these numbers are more on the order of 10^{15}.

The radio and TV frequency ranges are large also, which is why, for example, we can have many AM radio

[12]And in fact will probably be deliberately filtered out.

stations in a given city. The AM frequency range is divided into subranges, called **channels**. The width of these channels is on the order of the 4000 we need for a voice conversation. That means that the transmitter at a station needs to shift its content, which is something like in the [0,4000] range, to its channel range. It does that by multiplying its content times a sine wave of frequency equal to the center of the channel. If one applies a few trig identities, one finds that the product signal falls into the proper channel!

Accordingly, an optical fiber could also carry many simultaneous phone conversations.

Bandwidth also determines how fast we can set digital bits. Think of sending the sequence 10101010... If we graph this over time, we get a "squarewave" shape. Since it is repeating, it has a Fourier series. What happends if we double the bit rate? We get the same graph, only horizontally compressed by a factor of two. The effect of this on this graph's Fourier series is that, for example, our former a_3 will now be our new a_6, i.e. the $2\pi \cdot 3f_0$ frequency cosine wave component of the graph now has the double the old frequency, i.e. is now $2\pi \cdot 6f_0$. That in turn means that the effective bandwidth of our 10101010... signal has doubled too.

In other words: To send high bit rates, we need media with large bandwidths.