并行机器编程

Norm Matloff（著）
加州大学戴维斯分校

寇强（译）
印第安纳大学

GPU、多核、集群

目录

第 1 章　并行处理导论

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 of both the software and hardware.

1.1 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 the late 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 later at Pixar, whose graphics work requires extremely fast computers, he was always hoping someone would produce faster machines, so that he could 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.

We are now in the era of Big Data, which requires Big Computation, thus again generating a major need for parallel processing.

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.

1.1.3 Distributed Processing

In the above two subsections we've hit the two famous issues in computer science—time (speed) and space (memory capacity). But there is a third reason to do parallel processing, which actually has its own name, **distributed processing**. In a distributed database, for instance, parts of the database may be physically located in widely dispersed sites. If most transactions at a particular site arise locally, then we would make more efficient use of the network, and so on.

1.1.4 Our Focus Here

In this book, the primary emphasis is on processing speed.

1.2 Parallel Processing Hardware

This is a common scenario: Someone acquires a fancy new parallel machine, and excitedly writes a program to run on it—only to find that the parallel code is actually slower than the original serial version! This is due to lack of understanding of how the hardware works, at least at a high level.

This is not a hardware book, 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

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 **multicore** machines, in which two or more CPUs share a common memory,[1] are commonplace in the home and even in cell phones!

Multiprocessor Topologies

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

The multicore setup is effectively the same as SMP, except that the processors are all on one chip, attached to the bus.

So-called NUMA architectures will be discussed in Chapter **??**.

Memory Issues Etc.

Consider the SMP figure above.

- The Ps are processors, e.g. off-the-shelf chips such as Pentiums.

[1]The terminology gets confusing here. Although each core is a complete processor, people in the field tend to call the entire chip a "processor," referring to the cores, as, well, cores. In this book, the term *processor* will generally include cores, e.g. a dual-core chip will be considered to have two processors.

- 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 <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.

    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.

All of the above issues can have major on the speed of our program, as will be seen later.

1.2.2 Message-Passing Systems

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.

Example: Clusters

Here one has a set of commodity PCs and networks them for use as a parallel processing system. 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.

One factor which can be key to the success of a cluster 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 cluster context. A good network for a cluster is, for instance, Infiniband.

Clusters 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

1.3.1 Example: Matrix-Vector Multiply

To explain the 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.

In all the forms of parallelism, each node could be assigned some of the rows of A, and that node would multiply X by those rows, thus forming part of Y.

Note that in typical applications, the matrix A would be very large, say thousands of rows, possibly even millions. Otherwise the computation could be done quite satisfactorily in a **serial**, i.e. nonparallel manner, making parallel processing unnecessary..

1.3.2 Shared-Memory

Programmer View

In implementing the matrix-vector multiply example of Section 1.3.1 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.

Computation of the matrix-vector product AX would then involve the nodes somehow deciding which nodes will handle which rows of A. Each node would then multiply its assigned rows of A times X, and place the result directly in the proper section of the shared Y.

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,[2] Windows, Python, Java, Perl and now C++11 and R (via my Rdsm package) all support threaded programming.

In the typical implementation, a thread is a special case of an OS process. But the key 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

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

threads running in parallel.[3] Whenever a processor becomes available, the OS will assign some ready thread to it. So, among other things, this says that a thread might actually run on different processors during different turns.

Important note: Effective use of threads requires a basic understanding of how processes take turns executing. See Section **??** in the appendix of this book for this material.

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.

Example: Pthreads Prime Numbers Finder

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
```

[3] There may be other processes running too. So the threads of our program must still take turns with other processes running on the machine.

```c
// 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)  {
      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 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 says 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 threads, say if there are no other threads, or more generally if the number of other threads is less than or equal to the number of processors minus two. (Actually, the original thread is **main()**, but it lies dormant most of the time, as you'll see.)

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."

In most threaded programs, all communication between threads is done via global variables.[4] So even if you consider globals to be evil, they are a necessary evil in threads programming.

Personally I have always thought the stern admonitions against global variables are overblown anyway; see `http://heather.cs.ucdavis.edu/~matloff/globals.html`. But as mentioned, those admonitions are routinely ignored in threaded programming. For a nice discussion on this, see the paper by a famous MIT computer scientist on an Intel Web page, at `http://software.intel.com/en-us/articles/global-variable-reconsidered/?wapkw=%28parallelism%29`.

As mentioned earlier, the globals are shared by all processors.[5] 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
nextbase += 2
pthread_mutex_unlock(&nextbaselock);
```

[4] Technically one could use locals in **main()** (or whatever function it is where the threads are created) for this purpose, but this would be so unwieldy that it is seldom done.

[5] 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.

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 the code

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

at the same time. A common term used for this is that we wish the actions in the critical section to collectively be **atomic**, meaning not divisible among threads. 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, in order:

- 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 bottleneck 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.

Actually, we could have used Pthreads' built-in barrier function. We need to declare a barrier variable, e.g.

```
pthread_barrier_t barr;
```

and then call it like this:

```
pthread_barrier_wait(&barr);
```

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 **??**.

Role of the OS

Let's again ponder the role of the OS here. What happens when a thread tries to lock a lock:

- The lock call will ultimately cause a system call, causing the OS to run.
- The OS keeps track of the locked/unlocked status of each lock, so it will check that status.
- Say the lock is unlocked (a 0). Then the OS sets it to locked (a 1), and the lock call returns. The thread enters the critical section.
- When the thread is done, the unlock call unlocks the lock, similar to the locking actions.
- If the lock is locked at the time a thread makes a lock call, the call will block. The OS will mark this thread as waiting for the lock. When whatever thread currently using the critical section unlocks the lock, the OS will relock it and unblock the lock call of the waiting thread. If several threads are waiting, of course only one will be unblock.

Note that **main()** is a thread too, the original thread that spawns the others. However, it is dormant most of the time, due to its calls to **pthread_join()**.

Finally, keep in mind that although the globals variables are shared, the locals are not. Recall that local variables are stored on a stack. Each thread (just like each process in general) has its own stack. When a thread begins a turn, the OS prepares for this by pointing the stack pointer register to this thread's stack.

Debugging Threads Programs

Most debugging tools include facilities for threads. Here's an overview of how it works in GDB.

First, as you run a program under GDB, the creation of new threads will be announced, e.g.

```
(gdb) r 100 2
Starting program: /debug/primes 100 2
[New Thread 16384 (LWP 28653)]
[New Thread 32769 (LWP 28676)]
[New Thread 16386 (LWP 28677)]
[New Thread 32771 (LWP 28678)]
```

You can do backtrace (**bt**) etc. as usual. Here are some threads-related commands:

- `info threads` (gives information on all current threads)
- `thread 3` (change to thread 3)
- `break 88 thread 3` (stop execution when thread 3 reaches source line 88)
- `break 88 thread 3 if x==y` (stop execution when thread 3 reaches source line 88 and the variables x and y are equal)

Of course, many GUI IDEs use GDB internally, and thus provide the above facilities with a GUI wrapper. Examples are DDD, Eclipse and NetBeans.

Higher-Level Threads

The OpenMP library gives the programmer a higher-level view of threading. The threads are there, but rather hidden by higher-level abstractions. We will study OpenMP in detail in Chapter **??**, and use it frequently in the succeeding chapters, but below is an introductory example.

Example: Sampling Bucket Sort

This code implements the sampling bucket sort of Section **??**.

```
// OpenMP introductory example: sampling bucket sort

// compile: gcc -fopenmp -o bsort bucketsort.c

// set the number of threads via the environment variable
// OMP_NUM_THREADS, e.g. in the C shell

// setenv OMP_NUM_THREADS 8

#include <omp.h> // required
#include <stdlib.h>

// needed for call to qsort()
int cmpints(int *u, int *v)
{ if (*u < *v) return -1;
   if (*u > *v) return 1;
   return 0;
```

```c
}

// adds xi to the part array, increments npart, the length of part
void grab(int xi, int *part, int *npart)
{
    part[*npart] = xi;
    *npart += 1;
}

// finds the min and max in y, length ny,
// placing them in miny and maxy
void findminmax(int *y, int ny, int *miny, int *maxy)
{ int i,yi;
   *miny = *maxy = y[0];
   for (i = 1; i < ny; i++) {
      yi = y[i];
      if (yi < *miny) *miny = yi;
      else if (yi > *maxy) *maxy = yi;
   }
}

// sort the array x of length n
void bsort(int *x, int n)
{ // these are local to this function, but shared among the threads
   float *bdries; int *counts;
   #pragma omp parallel
   // entering this block activates the threads, each executing it
   { // variables declared below are local to each thread
      int me = omp_get_thread_num();
      // have to do the next call within the block, while the threads
      // are active
      int nth = omp_get_num_threads();
      int i,xi,minx,maxx,start;
      int *mypart;
      float increm;
      int SAMPLESIZE;
    // now determine the bucket boundaries; nth - 1 of them, by
    // sampling the array to get an idea of its range
    #pragma omp single // only 1 thread does this, implied barrier at
       end
    {
      if (n > 1000) SAMPLESIZE = 1000;
      else SAMPLESIZE = n / 2;
      findminmax(x,SAMPLESIZE,&minx,&maxx);
      bdries = malloc((nth-1)*sizeof(float));
      increm = (maxx - minx) / (float) nth;
      for (i = 0; i < nth-1; i++)
        bdries[i] = minx + (i+1) * increm;
      // array to serve as the count of the numbers of elements of x
      // in each bucket
      counts = malloc(nth*sizeof(int));
    }
    // now have this thread grab its portion of the array; thread 0
    // takes everything below bdries[0], thread 1 everything between
    // bdries[0] and bdries[1], etc., with thread nth-1 taking
    // everything over bdries[nth-1]
    mypart = malloc(n*sizeof(int)); int nummypart = 0;
    for (i = 0; i < n; i++) {
      if (me == 0) {
        if (x[i] <= bdries[0]) grab(x[i],mypart,&nummypart);
      }
      else if (me < nth-1) {
        if (x[i] > bdries[me-1] && x[i] <= bdries[me])
          grab(x[i],mypart,&nummypart);
      } else
        if (x[i] > bdries[me-1]) grab(x[i],mypart,&nummypart);
    }
    // now record how many this thread got
    counts[me] = nummypart;
    // sort my part
    qsort(mypart,nummypart,sizeof(int),cmpints);
    #pragma omp barrier // other threads need to know all of counts
```

```
      // copy sorted chunk back to the original array; first find start
         point
      start = 0;
      for (i = 0; i < me; i++) start += counts[i];
      for (i = 0; i < nummypart; i++) {
         x[start+i] = mypart[i];
      }
   }
   // implied barrier here; main thread won't resume until all threads
   // are done
}

int main(int argc, char **argv)
{
   // test case
   int n = atoi(argv[1]), *x = malloc(n*sizeof(int));
   int i;
   for (i = 0; i < n; i++) x[i] = rand() % 50;
   if (n < 100)
      for (i = 0; i < n; i++) printf("%d\n",x[i]);
   bsort(x,n);
   if (n <= 100) {
      printf("x after sorting:\n");
      for (i = 0; i < n; i++) printf("%d\n",x[i]);
   }
}
```

Details on OpenMP are presented in Chapter **??**. Here is an overview of a few of the OpenMP constructs available:

- **#pragma omp for**
  In our example above, we wrote our own code to assign specific threads to do specific parts of the work. An alternative is to write an ordinary **for** loop that iterates over all the work to be done, and then ask OpenMP to assign specific iterations to specific threads. To do this, insert the above pragma just before the loop.
- **#pragma omp critical**

The block that follows is implemented as a critical section. OpenMP sets up the locks etc. for you, alleviating you of work and alleviating your code of clutter.

Debugging OpenMP

Since there are threads underlying the OpenMP execution, you should be able to use your debugging tool's threads facilities. Note, though, that this may not work perfectly well.

Some versions of GCC/GDB, for instance, do not display some local variables. Let's consider two categories of such variables:

(a) Variables within a **parallel** block, such as **me** in **bsort()** in Section 1.3.2.

(b) Variables that are not in a **parallel** block, but which are still local to a function that contains such a block. An example is **counts** in **bsort()**.

You may find that when you try to use GDB's **print** command, GDB says there is no such variable.

The problem seems to arise from a combination of (i) optimzation, so that a variable is placed in a register and basically eliminated from the namespace, and (ii) some compilers implement OpenMP by actually making special versions of the function being debugged.

In GDB, one possible workaround is to use the **-gstabs+** option when compiling, instead of **-g**. But here is a more general workarounds. Let's consider variables of type (b) first.

The solution is to temporarily change these variables to globals, e.g.

```
int *counts;
void bsort(int *x, int n)
```

This would still be all right in terms of program correctness, because the variables in (b) are global to the threads anyway. (Of course, make sure not to have another global of the same name!) The switch would only be temporary, during debugging, to be switched back later so that in the end **bsort()** is self-contained.

The same solution works for category (a) variables, with an added line:

```
int me;
#pragma omp threadprivate(me)
void bsort(int *x, int n)
```

What this does is make separate copies of **me** as global variables, one for each thread. As globals, GCC won't engage in any shenanigans with them. :-) One does have to keep in mind that they will retain there values upon exit from a parallel block etc., but the workaround does work.

1.3.3 Message Passing

Programmer View

Again consider the matrix-vector multiply example of Section 1.3.1. In contrast to the shared-memory case, in the message-passing paradigm all nodes would have separate copies of A, X and Y. Our example in Section 1.3.2 would now change. 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.

To compute the matrix-vector product, then, would involve the following. One node, say node 0, would distribute the rows of A to the various other nodes. Each node would receive a different set of rows. The vector X would be sent to all nodes.[6] Each node would then multiply X by the node's assigned rows of A, and then send the result back to node 0. The latter would collect those results, and store them in Y.

Example: MPI Prime Numbers Finder

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

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
```

[6] In a more refined version, X would be parceled out to the nodes, just as the rows of A are.

```

   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; node 2
   then removes the multiples of 7, and so on; the last node must check
   whatever is left

   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 */

#include <mpi.h>  // mandatory

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

void Init(int Argc,char **Argv)
{  int DebugWait;
   N = atoi(Argv[1]);
   // start debugging section
   DebugWait = atoi(Argv[2]);
   while (DebugWait) ;  // deliberate infinite loop; see below
   /* the above loop is here to synchronize all nodes for debugging;
      if DebugWait is specified as 1 on the mpirun command line, all
      nodes wait here until the debugging programmer starts GDB at
      all nodes (via attaching to OS process number), then sets
      some breakpoints, then GDB sets DebugWait to 0 to proceed; */
   // end debugging section
   MPI_Init(&Argc,&Argv);  // mandatory to begin any MPI program
   // 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 == NNodes-1) T1 = MPI_Wtime();
}

void Node0()
{  int I,ToCheck,Dummy,Error;
   for (I = 1; I <= N/2; I++)  {
      ToCheck = 2 * I + 1;  // latest number to check for div3
      if (ToCheck > N) break;
      if (ToCheck % 3 > 0)  // not divis by 3, so send it down the pipe
         // send the string at ToCheck, consisting of 1 MPI integer, to
         // node 1 among MPI_COMM_WORLD, with a message type PIPE_MSG
         Error = MPI_Send(&ToCheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
         // error not checked in this code
   }
   // sentinel
   MPI_Send(&Dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
}

void NodeBetween()
{  int ToCheck,Dummy,Divisor;
   MPI_Status Status;
   // first received item gives us our prime divisor
   // receive into Divisor 1 MPI integer from node Me-1, of any message
   // type, and put information about the message in Status
   MPI_Recv(&Divisor,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
   while (1)  {
      MPI_Recv(&ToCheck,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
      // if the message type was END_MSG, end loop
      if (Status.MPI_TAG == END_MSG) break;
      if (ToCheck % Divisor > 0)
         MPI_Send(&ToCheck,1,MPI_INT,Me+1,PIPE_MSG,MPI_COMM_WORLD);
   }
   MPI_Send(&Dummy,1,MPI_INT,Me+1,END_MSG,MPI_COMM_WORLD);
}

NodeEnd()
```

```
{  int ToCheck,PrimeCount,I,IsComposite,StartDivisor;
   MPI_Status Status;
   MPI_Recv(&StartDivisor,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
   PrimeCount = Me + 2;  /* must account for the previous primes, which
                            won't be detected below */
   while (1)  {
      MPI_Recv(&ToCheck,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
      if (Status.MPI_TAG == END_MSG) break;
      IsComposite = 0;
      for (I = StartDivisor; 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);
}

int main(int argc,char **argv)
{  Init(argc,argv);
   // all nodes run this same program, but different nodes take
   // different actions
   if (Me == 0) Node0();
   else if (Me == NNodes-1) NodeEnd();
        else NodeBetween();
   // 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.

1.3.4 Scatter/Gather

Technically, the **scatter/gather** programmer world view is a special case of message passing. However, it has become so pervasive as to merit its own section here.

In this paradigm, one node, say node 0, serves as a **manager**, while the others serve as **workers**. The parcels out work to the workers, who process their respective chunks of the data and return the results to the manager. The latter receives the results and combines them into the final product.

The matrix-vector multiply example in Section 1.3.3 is an example of scatter/gather.

As noted, scatter/gather is very popular. Here are some examples of packages that use it:

- MPI includes scatter and gather functions (Section **??**).
- Hadoop/MapReduce Computing (Chapter **??**) is basically a scatter/gather operation.
- The **snow** package (Section 1.3.4) for the R language is also a scatter/gather operation.

R snow Package

Base R does not include parallel processing facilities, but includes the **parallel** library for this purpose, and a number of other parallel libraries are available as well. The **parallel** package arose from the merger (and slight modifcation) of two former user-contributed libraries, **snow** and **multicore**. The former (and essentially the latter) uses the scatter/-gather paradigm, and so will be introduced in this section; see Section 1.3.4 for further details. for convenience, I'll refer to the portion of **parallel** that came from **snow** simply as **snow**.

Let’s use matrix-vector multiply as an example to learn from:

```
> library(parallel)
> c2 <- makePSOCKcluster(rep("localhost",2))
> c2
socket cluster with 2 nodes on host  ‘’ localhost
> mmul
function(cls,u,v) {
   rowgrps <- splitIndices(nrow(u),length(cls))
   grpmul <- function(grp) u[grp,] %*% v
   mout <- clusterApply(cls,rowgrps,grpmul)
   Reduce(c,mout)
}
> a <- matrix(sample(1:50,16,replace=T),ncol=2)
> a
     [,1] [,2]
[1,] 34 41
[2,] 10 28
[3,] 44 23
[4,] 7 29
[5,] 6 24
[6,] 28 29
[7,] 21 1
[8,] 38 30
> b <- c(5,-2)
> b
[1] 5 -2
> a %*% b # serial multiply
     [,1]
[1,] 88
[2,] -6
[3,] 174
[4,] -23
[5,] -18
[6,] 82
[7,] 103
```

```
[8,] 130
> clusterExport(c2,c("a","b")) # send a,b to workers
> clusterEvalQ(c2,a) # check that they have it
[[1]]
     [,1] [,2]
[1,] 34 41
[2,] 10 28
[3,] 44 23
[4,] 7 29
[5,] 6 24
[6,] 28 29
[7,] 21 1
[8,] 38 30

[[2]]
     [,1] [,2]
[1,] 34 41
[2,] 10 28
[3,] 44 23
[4,] 7 29
[5,] 6 24
[6,] 28 29
[7,] 21 1
[8,] 38 30
> mmul(c2,a,b) # test our parallel code
[1] 88 -6 174 -23 -18 82 103 130
```

What just happened?

First we set up a **snow** cluster. The term should not be confused with hardware systems we referred to as "clusters" earlier. We are simply setting up a group of R processes that will communicate with each other via TCP/IP sockets.

In this case, my cluster consists of two R processes running on the machine from which I invoked **makePSOCKcluster()**. (In TCP/IP terminology, **localhost** refers to the local machine.) If I were to run the Unix **ps** command, with appropriate options, say **ax**, I'd see three R processes. I saved the cluster in **c2**.

On the other hand, my **snow** cluster could indeed be set up on a real cluster, e.g.

```
c3 <- makePSOCKcluster(c("pc28","pc29","pc29"))
```

where **pc28** etc. are machine names.

In preparing to test my parallel code, I needed to ship my matrices **a** and **b** to the workers:

```
> clusterExport(c2,c("a","b")) # send a,b to workers
```

Note that this function assumes that **a** and **b** are global variables at the invoking node, i.e. the manager, and it will place copies of them in the global workspace of the worker nodes.

Note that the copies are independent of the originals; if a worker changes, say, **b[3]**, that change won't be made at the manager or at the other worker. This is a message-passing system, indeed.

So, how does the **mmul** code work? Here's a handy copy:

```
mmul <- function(cls,u,v) {
   rowgrps <- splitIndices(nrow(u),length(cls))
   grpmul <- function(grp) u[grp,] %*% v
   mout <- clusterApply(cls,rowgrps,grpmul)
   Reduce(c,mout)
}
```

As discussed in Section 1.3.1, our strategy will be to partition the rows of the matrix, and then have different workers handle different groups of rows. Our call to **splitIndices()** sets this up for us.

That function does what its name implies, e.g.

```
> splitIndices(12,5)
[[1]]
[1] 1 2 3

[[2]]
[1] 4 5

[[3]]
[1] 6 7

[[4]]
```

```
[1] 8 9

[[5]]
[1] 10 11 12
```

Here we asked the function to partition the numbers 1,...,12 into 5 groups, as equal-sized as possible, which you can see is what it did. Note that the type of the return value is an R list.

So, after executing that function in our **mmul()** code, **rowgrps** will be an R list consisting of a partitioning of the row numbers of **u**, exactly what we need.

The call to **clusterApply()** is then where the actual work is assigned to the workers. The code

```
mout <- clusterApply(cls,rowgrps,grpmul)
```

instructs **snow** to have the first worker process the rows in **rowgrps**[[**1**]], the second worker to work on **rowgrps**[[**2**]], and so on. The **clusterApply()** function expects its second argument to be an R list, which is the case here.

Each worker will then multiply **v** by its row group, and return the product to the manager. However, the product will again be a list, one component for each worker, so we need **Reduce()** to string everything back together.

Note that R does allow functions defined within functions, which the locals and arguments of the outer function becoming global to the inner function.

Note that **a** here could have been huge, in which case the export action could slow down our program. If **a** were not needed at the workers other than for this one-time matrix multiply, we may wish to change to code so that we send each worker only the rows of **a** that we need:

```
mmul1 <- function(cls,u,v) {
   rowgrps <- splitIndices(nrow(u),length(cls))
   uchunks <- Map(function(grp) u[grp,],rowgrps)
   mulchunk <- function(uc) uc %*% v
   mout <- clusterApply(cls,uchunks,mulchunk)
   Reduce(c,mout)
}
```

Let's test it:

```
> a <- matrix(sample(1:50,16,replace=T),ncol=2)
> b <- c(5,-2)
> clusterExport(c2,"b") # don't send a
 a
     [,1] [,2]
[1,] 10 26
[2,] 1 34
[3,] 49 30
[4,] 39 41
[5,] 12 14
[6,] 2 30
[7,] 33 23
[8,] 44 5
> a %*% b
     [,1]
[1,] -2
[2,] -63
[3,] 185
[4,] 113
[5,] 32
[6,] -50
[7,] 119
[8,] 210
> mmul1(c2,a,b)
[1] -2 -63 185 113 32 -50 119 210
```

Note that we did not need to use **clusterExport()** to send the chunks of **a** to the workers, as the call to **clusterApply()** does this, since it sends the arguments,