并行机器编程

Norm Matloff（著）
加州大学戴维斯分校

寇强（译）
华南统计科学研究中心
印第安纳大学
新浪微博：Gossip_useR

GPU、多核、集群

http://heather.cs.ucdavis.edu/~matloff/probstatbook.html
本书更新网址：

英文版：http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf
中文版：https://github.com/thirdwing/ParaBook

作者和译者尽了最大努力，但书中错误在所难免
如果对翻译有任何疑问，请随时通过邮件kouqiang@mail3.sysu.edu.cn联系我

关于本书

为什么本书和其它的并行编程书籍不同呢？原因在于我们主要关注在实现层面：

- 这里几乎没有理论内容，诸如 O() 分析、最大理论加速、PRAM、有向无环图 (DAG) 等等。
- 书中使用的都是真实代码。
- 我们使用的都是主流的并行平台，包括 OpenMP、CUDA 和 MPI，而没有使用其它仍处于实验阶段的语言。
- 关于性能的主题——通信延迟、内存/网络连接、负载均衡等，在全书中交叉进行，并且都是在特定平台或应用的层面进行讨论的。
- 相当关注调试技术。

书中使用的主要编程语言 C/C++，但也使用了一些 R 代码。R 已经是最主流的用于数据分析的语言。作为一门脚本语言，R 可以用于快速原型构建。在本书中，我用 R 将可以一些例子表述得远远比用 C/C++ 要简洁，从而使得学生更容易的理解所使用的并行计算原则。出于同样的原因，学生们也可以更容易地编写并行代码，更加集中精力于这些原则之上。另外，R 也有相当丰富的并行库。

我们假设学生在编程方面是有相当经验的，并有包括线性代数在内的数学背景。附录里回顾了本书所需要的数学知识。另一个附录提供了不同系统问题的概述，包括进程调度和虚拟内存等。

需要特别说明的是，书中多数代码没有进行优化。我们主要关注的是技术和语言使用的清晰明了。然而，有很多影响速度的因素值得讨论，比如高速缓存一致性问题、网络延迟、GPU 内存结构等等。

这里展示了你可以如何使用书中的代码：本书使用 LaTeX 排版，原始的**.tex** 文件可以在 `http://heather.cs.ucdavis.edu/~matloff/158/PLN`下载。请直接下载相关文件（文件名应该足够明确），之后使用一个文本编辑器进行裁剪，从而得到感兴趣的代码。

为了向学生展示研究和教学的相互促进关系，我会时不时引用我的一些研究工作。

如同我的其它开源图书，本书是在**不停变动**中的。我会继续添加新的主题、新的示例等等，当然也会修补漏洞和改善说明。由于这个原因，所以保存本书最新版本的链接，`http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`，比保存一份拷贝更好。

同样出于这个原因，我非常希望得到反馈。这里我希望感谢 Stuart Ambler、Matt Butner、Stuart Hansen、Bill Hsu、Sameer Khan、Mikel McDaniel、Richard Minner、Lars Seeman、Marc Sosnick 和 Johan Wikström 的评论。特别感谢 Hsu 教授为我提供了高级的 GPU 设备。

各位可能对我另一本关于概览和统计的开源图书感兴趣，可以在 `http://heather.cs.ucdavis.edu/probstatbook`下载。

作者简介

Norm Matloff 博士是加州大学戴维斯分校计算机教授，曾任同校的统计学教授。他曾是硅谷的一名数据库软件开发人员，也曾作为统计咨询师为 Kaiser Permanente Health Plan 等公司工作过。

Matloff 博士生于 Los Angeles，在 East Los Angeles 和 San Gabriel Valley 长大。他从 UCLA 得到了纯数学的博士学位，研究方向为概率论和统计他在计算机科学和统计学方面发表了大量论文，现在的研究方向是并行处理、统计学和回归方法。

Matloff 博士曾是 UNESCO 下的数据库安全国际委员会，IFIP Working Group 11.3 成员。他是戴维斯分校统计系的创始人之一，并参与了计算机系的建立。他在戴维斯分校被授予 Distinguished Teaching Award and Distinguished Public Service Award。

Matloff 博士是两本书的作者，并编写了大量广泛使用的网络教程，涉及 Linux 和 python 语言。他和 Peter Salzman 博士是《软件调试的艺术》[①]一书的作者。Matloff 博士关于 R 语言的《R 语言编程艺术》[②]一书已于 2011 年出版。他的新书，*Parallel Computation for Data Science* 会于 2014 年出版。他还写作了很多开源图书，包括 *From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science*（`http://heather.cs.ucdavis.edu/probstatbook`），和《并行机器编程》（`http://heather.cs.ucdavis.edu/~matloff/ParProcBook.pdf`）。

[①]译者注：The Art of Debugging with GDB, DDD, and Eclipse，中文版已由人民邮电出版社出版
[②]译者注：The Art of R Programming，中文版已由机械工业出版社出版

目录

第 1 章　并行处理入门

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 3.

Memory Issues Etc.

Consider the SMP figure above.

- 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.

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

[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.

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

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

and then node 15 were to subsequently execute

```
1  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 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.

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

[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.

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.

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

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

```
95        if (prime[i])  {
96           nprimes++;
97        }
98     printf("the number of primes found was %d\n",nprimes);
99
100 }
```

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 3.

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 4, 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 13.5.

```
// 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;
```

```
16    if (*u > *v) return 1;
17    return 0;
18 }
19
20 // adds xi to the part array, increments npart, the length of part
21 void grab(int xi, int *part, int *npart)
22 {
23     part[*npart] = xi;
24     *npart += 1;
25 }
26
27 // finds the min and max in y, length ny,
28 // placing them in miny and maxy
29 void findminmax(int *y, int ny, int *miny, int *maxy)
30 { int i,yi;
31    *miny = *maxy = y[0];
32    for (i = 1; i < ny; i++) {
33       yi = y[i];
34       if (yi < *miny) *miny = yi;
35       else if (yi > *maxy) *maxy = yi;
36    }
37 }
38
39 // sort the array x of length n
40 void bsort(int *x, int n)
41 { // these are local to this function, but shared among the threads
42    float *bdries; int *counts;
43    #pragma omp parallel
44    // entering this block activates the threads, each executing it
45    { // variables declared below are local to each thread
46       int me = omp_get_thread_num();
47       // have to do the next call within the block, while the threads
48       // are active
49       int nth = omp_get_num_threads();
50       int i,xi,minx,maxx,start;
51       int *mypart;
52       float increm;
53       int SAMPLESIZE;
54       // now determine the bucket boundaries; nth - 1 of them, by
55       // sampling the array to get an idea of its range
56       #pragma omp single // only 1 thread does this, implied barrier at end
57       {
58          if (n > 1000) SAMPLESIZE = 1000;
```

```
59          else SAMPLESIZE = n / 2;
60          findminmax(x,SAMPLESIZE,&minx,&maxx);
61          bdries = malloc((nth-1)*sizeof(float));
62          increm = (maxx - minx) / (float) nth;
63          for (i = 0; i < nth-1; i++)
64             bdries[i] = minx + (i+1) * increm;
65          // array to serve as the count of the numbers of elements of x
66          // in each bucket
67          counts = malloc(nth*sizeof(int));
68       }
69       // now have this thread grab its portion of the array; thread 0
70       // takes everything below bdries[0], thread 1 everything between
71       // bdries[0] and bdries[1], etc., with thread nth-1 taking
72       // everything over bdries[nth-1]
73       mypart = malloc(n*sizeof(int)); int nummypart = 0;
74       for (i = 0; i < n; i++) {
75          if (me == 0) {
76             if (x[i] <= bdries[0]) grab(x[i],mypart,&nummypart);
77          }
78          else if (me < nth-1) {
79             if (x[i] > bdries[me-1] && x[i] <= bdries[me])
80                grab(x[i],mypart,&nummypart);
81          } else
82             if (x[i] > bdries[me-1]) grab(x[i],mypart,&nummypart);
83       }
84       // now record how many this thread got
85       counts[me] = nummypart;
86       // sort my part
87       qsort(mypart,nummypart,sizeof(int),cmpints);
88       #pragma omp barrier // other threads need to know all of counts
89       // copy sorted chunk back to the original array; first find start point
90       start = 0;
91       for (i = 0; i < me; i++) start += counts[i];
92       for (i = 0; i < nummypart; i++) {
93          x[start+i] = mypart[i];
94       }
95    }
96    // implied barrier here; main thread won't resume until all threads
97    // are done
98 }
99
100 int main(int argc, char **argv)
101 {
```

```
102    // test case
103    int n = atoi(argv[1]), *x = malloc(n*sizeof(int));
104    int i;
105    for (i = 0; i < n; i++) x[i] = rand() % 50;
106    if (n < 100)
107       for (i = 0; i < n; i++) printf("%d\n",x[i]);
108    bsort(x,n);
109    if (n <= 100) {
110       printf("x after sorting:\n");
111       for (i = 0; i < n; i++) printf("%d\n",x[i]);
112    }
113 }
```

Details on OpenMP are presented in Chapter 4. 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.

```
1 int *counts;
```

```
2 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:

```
1 int me;
2 #pragma omp threadprivate(me)
3 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

```
1    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.

```
1  /* MPI sample program; NOT INTENDED TO BE EFFICIENT as a prime
2     finder, either in algorithm or implementation
3
4     MPI (Message Passing Interface) is a popular package using
```

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

```
5     the "message passing" paradigm for communicating between
6     processors in parallel applications; as the name implies,
7     processors communicate by passing messages using "send" and
8     "receive" functions
9
10    finds and reports the number of primes less than or equal to N
11
12    uses a pipeline approach:  node 0 looks at all the odd numbers (i.e.
13    has already done filtering out of multiples of 2) and filters out
14    those that are multiples of 3, passing the rest to node 1; node 1
15    filters out the multiples of 5, passing the rest to node 2; node 2
16    then removes the multiples of 7, and so on; the last node must check
17    whatever is left
18
19    note that we should NOT have a node run through all numbers
20    before passing them on to the next node, since we would then
21    have no parallelism at all; on the other hand, passing on just
22    one number at a time isn't efficient either, due to the high
23    overhead of sending a message if it is a network (tens of
24    microseconds until the first bit reaches the wire, due to
25    software delay); thus efficiency would be greatly improved if
26    each node saved up a chunk of numbers before passing them to
27    the next node */
28
29 #include <mpi.h>  // mandatory
30
31 #define PIPE_MSG 0  // type of message containing a number to be checked
32 #define END_MSG 1  // type of message indicating no more data will be coming
33
34 int NNodes,  // number of nodes in computation
35     N,  // find all primes from 2 to N
36     Me;  // my node number
37 double T1,T2;  // start and finish times
38
39 void Init(int Argc,char **Argv)
40 {  int DebugWait;
41    N = atoi(Argv[1]);
42    // start debugging section
43    DebugWait = atoi(Argv[2]);
44    while (DebugWait) ;  // deliberate infinite loop; see below
45    /* the above loop is here to synchronize all nodes for debugging;
46       if DebugWait is specified as 1 on the mpirun command line, all
47       nodes wait here until the debugging programmer starts GDB at
48       all nodes (via attaching to OS process number), then sets
49       some breakpoints, then GDB sets DebugWait to 0 to proceed; */
50    // end debugging section
51    MPI_Init(&Argc,&Argv);  // mandatory to begin any MPI program
52    // puts the number of nodes in NNodes
53    MPI_Comm_size(MPI_COMM_WORLD,&NNodes);
54    // puts the node number of this node in Me
```

```
55    MPI_Comm_rank(MPI_COMM_WORLD,&Me);
56    // OK, get started; first record current time in T1
57    if (Me == NNodes-1) T1 = MPI_Wtime();
58 }
59
60 void Node0()
61 {  int I,ToCheck,Dummy,Error;
62    for (I = 1; I <= N/2; I++)  {
63       ToCheck = 2 * I + 1;  // latest number to check for div3
64       if (ToCheck > N) break;
65       if (ToCheck % 3 > 0)  // not divis by 3, so send it down the pipe
66          // send the string at ToCheck, consisting of 1 MPI integer, to
67          // node 1 among MPI_COMM_WORLD, with a message type PIPE_MSG
68          Error = MPI_Send(&ToCheck,1,MPI_INT,1,PIPE_MSG,MPI_COMM_WORLD);
69          // error not checked in this code
70    }
71    // sentinel
72    MPI_Send(&Dummy,1,MPI_INT,1,END_MSG,MPI_COMM_WORLD);
73 }
74
75 void NodeBetween()
76 {  int ToCheck,Dummy,Divisor;
77    MPI_Status Status;
78    // first received item gives us our prime divisor
79    // receive into Divisor 1 MPI integer from node Me-1, of any message
80    // type, and put information about the message in Status
81    MPI_Recv(&Divisor,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
82    while (1)  {
83       MPI_Recv(&ToCheck,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
84       // if the message type was END_MSG, end loop
85       if (Status.MPI_TAG == END_MSG) break;
86       if (ToCheck % Divisor > 0)
87          MPI_Send(&ToCheck,1,MPI_INT,Me+1,PIPE_MSG,MPI_COMM_WORLD);
88    }
89    MPI_Send(&Dummy,1,MPI_INT,Me+1,END_MSG,MPI_COMM_WORLD);
90 }
91
92 NodeEnd()
93 {  int ToCheck,PrimeCount,I,IsComposite,StartDivisor;
94    MPI_Status Status;
95    MPI_Recv(&StartDivisor,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
96    PrimeCount = Me + 2;  /* must account for the previous primes, which
97                             won't be detected below */
98    while (1)  {
99       MPI_Recv(&ToCheck,1,MPI_INT,Me-1,MPI_ANY_TAG,MPI_COMM_WORLD,&Status);
100      if (Status.MPI_TAG == END_MSG) break;
101      IsComposite = 0;
102      for (I = StartDivisor; I*I <= ToCheck; I += 2)
103         if (ToCheck % I == 0)  {
104            IsComposite = 1;
```

```
105             break;
106          }
107       if (!IsComposite) PrimeCount++;
108    }
109    /* check the time again, and subtract to find run time */
110    T2 = MPI_Wtime();
111    printf("elapsed time = %f\n",(float)(T2-T1));
112    /* print results */
113    printf("number of primes = %d\n",PrimeCount);
114 }
115
116 int main(int argc,char **argv)
117 {  Init(argc,argv);
118    // all nodes run this same program, but different nodes take
119    // different actions
120    if (Me == 0) Node0();
121    else if (Me == NNodes-1) NodeEnd();
122         else NodeBetween();
123    // mandatory for all MPI programs
124    MPI_Finalize();
125 }
126
127 /* explanation of "number of items" and "status" arguments at the end
128    of MPI_Recv():
129
130    when receiving a message you must anticipate the longest possible
131    message, but the actual received message may be much shorter than
132    this; you can call the MPI_Get_count() function on the status
133    argument to find out how many items were actually received
134
135    the status argument will be a pointer to a struct, containing the
136    node number, message type and error status of the received
137    message
138
139    say our last parameter is Status; then Status.MPI_SOURCE
140    will contain the number of the sending node, and
141    Status.MPI_TAG will contain the message type; these are
142    important if used MPI_ANY_SOURCE or MPI_ANY_TAG in our
143    node or tag fields but still have to know who sent the
144    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 7.4).
- Hadoop/MapReduce Computing (Chapter 9) 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,

第 2 章　Recurring Performance Issues

2.1　通信瓶颈

2.2　负载均衡

2.3　易并行应用

2.3.1　“易并行” 究竟意味着什么?

2.3.2　迭代算法

2.4　静态（可能随机）的任务分配要优于动态

2.4.1　1

2.4.2　2

2.4.3　3

2.5　延迟和带宽

2.6　Relative Merits：内存共享系统和信息传递系统的性能比较

2.7　内存分配问题

2.8　内存共享系统的特别问题

第 3 章　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.

Note: To get the most of this section—which is used frequently in the rest of this book—you may wish to read the material on array storage in the appendix of this book, Section **??**.

3.1 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

```
1 movl 200, %ebx
```

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. (Note, however, that each CPU has a separate register set, so each will have its own independent EAX.) 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.

3.2 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** or **banks**. This way several memory accesses can be done simultaneously.

In this section, assume for simplicity that our machine has 32-bit words. This is still true for many GPUs, in spite of the widespread use of 64-bit general-purpose machines today, and in any case, the numbers here can easily be converted to the 64-bit case.

Note that this means that consecutive words differ in address by 4. Let's thus define the word-address of a word to be its ordinary address divided by 4. Note that this is also its address with the lowest two bits deleted.

3.2.1 Interleaving

There is a question of how to divide up the memory into banks. There are two main ways to do this:

(a) **High-order interleaving:** Here consecutive words are in the same bank (except at boundaries). For example, suppose for simplicity that our memory consists of word-addresses 0 through 1023, and that there are four banks, M0 through M3. Then M0 would contain word-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 banks (except when we get to the right end). In the example above, if we used low-order interleaving, then word-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 have eight banks. Then under high-order interleaving, the first three bits of a word-address would be taken to be the bank number, with the remaining bits being address within bank. Under low-order interleaving, the three least significant bits would be used to determine bank number.

Low-order interleaving has often been 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 elements of these vectors are spread across the various banks, so fast access is possible.

A more modern use of low-order interleaving, but with the same motivation as with the vector processors, is in GPUs (Chapter 5).

High-order interleaving might work well in matrix applications, for instance, where we can partition the matrix into blocks, and have different processors work on different blocks. In image processing applications, we can have different processors work on different parts of the image. Such partitioning almost never works perfectly—e.g. computation for one part of an image may need information from another part—but if we are careful we can get good results.

3.2.2 Bank Conflicts and Solutions

Consider an array **x** of 16 million elements, whose sum we wish to compute, say using 16 threads. Suppose we have four memory banks, with low-order interleaving.

A naive implementation of the summing code might be

```
parallel for thr = 0 to 15
   localsum = 0
   for j = 0 to 999999
      localsum += x[thr*1000000+j]
   grandsum += localsumsum
```

In other words, thread 0 would sum the first million elements, thread 1 would sum the second million, and so on. After summing its portion of the array, a thread would then add its sum to a grand total. (The threads *could* of course add to **grandsum** directly in each iteration of the loop, but this would cause too much traffic to memory, thus causing slowdowns.)

Suppose for simplicity that there is one address per word (it is usually one address per byte).

Suppose also for simplicity that the threads run in lockstep, so that they all attempt to access memory at once. On a multicore/multiprocessor machine, this may not occur, but it in fact typically *will* occur in a GPU setting.

A problem then arises. To make matters simple, suppose that **x** starts at an address that is a multiple of 4, thus in bank 0. (The reader should think about how to adjust this to the other three cases.) On the very first memory access, thread 0 accesses **x[0]** in bank 0, thread 1 accesses **x[1000000]**, also in bank 0, and so on—and *these will all be in memory bank 0*! Thus there will be major conflicts, hence major slowdown.

A better approach might be to have any given thread work on every sixteenth element of **x**, instead of on contiguous elements. Thread 0 would work on **x[1000000]**, **x[1000016]**, **x[10000032**,...; thread 1 would handle **x[1000001]**, **x[1000017]**, **x[10000033**,...; and so on:

```
parallel for thr = 0 to 15
   localsum = 0
   for j = 0 to 999999
      localsum += x[16*j+thr]
   grandsum += localsumsum
```

Here, consecutive threads work on consecutive elements in **x**.[①] That puts them in separate banks, thus no conflicts, hence speedy performance.

In general, avoiding bank conflicts is an art, but there are a couple of approaches we can try.

- We can rewrite our algorithm, e.g. use the second version of the above code instead of the first.

[①] Here thread 0 is considered "consecutive" to thread 15, in a wraparound manner.

- We can add **padding** to the array. For instance in the first version of our code above, we could lengthen the array from 16 million to 16000016, placing padding in words 1000000, 2000001 and so on. We'd tweak our array indices in our code accordingly, and eliminate bank conflicts that way.

In the first approach above, the concept of **stride** often arises. It is defined to be the distance betwwen array elements in consecutive accesses by a thread. In our original code to compute **grandsum**, the stride was 1, since each array element accessed by a thread is 1 past the last access by that thread. In our second version, the stride was 16.

Strides of greater than 1 often arise in code that deals with multidimensional arrays. Say for example we have two-dimensional array with 16 columns. In C/C++, which uses row-major order, access of an entire column will have a stride of 16. Access down the main diagonal will have a stride of 17.

Suppose we have b banks, again with low-order interleaving. You should experiment a bit to see that an array access with a stride of s will access s different banks if and only if s and b are relatively prime, i.e. the greatest common divisor of s and b is 1. This can be proven with group theory.

Another strategy, useful for collections of complex objects, is to set up **structs of arrays** rather than **arrays of structs**. Say for instance we are working with data on workers, storing for each worker his name, salary and number of years with the firm. We might naturally write code like this:

```
struct {
   char name[25];
   float salary;
   float yrs;
} x[100];
```

That gives a 100 structs for 100 workers. Again, this is very natural, but it may make for poor memory access patterns. Salary values for the various workers will no longer be contiguous, for instance, even though the **struct**s are contiguous. This could cause excessive cache misses.

One solution would be to add padding to each **struct**, so that the salary values are a word apart in memory. But another approach would be to replace the above arrays of **struct**s by a **struct** of arrays:

```
struct {
   char *name[]100;
   float salary[100];
   float yrs[100];
}
```

3.2.3 Example: Code to Implement Padding

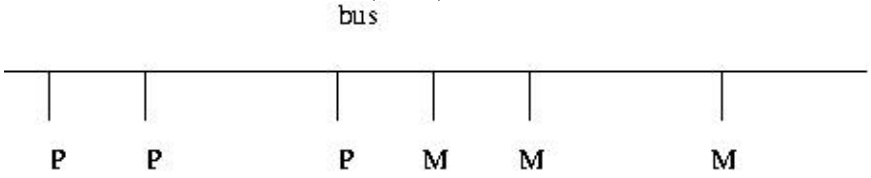
As discussed above, array padding is used to try to get better parallel access to memory banks. The code below is aimed to provide utilities to assist in this. Details are explained in the comments.

```
1
2  // routines to initialize, read and write
3  // padded versions of a matrix of floats;
4  // the matrix is nominally mxn, but its
5  // rows will be padded on the right ends,
6  // so as to enable a stride of s down each
7  // column; it is assumed that s >= n
8
9  // allocate space for the padded matrix,
10 // initially empty
11 float *padmalloc(int m, int n, int s) {
12   return(malloc(m*s*sizeof(float)));
13 }
14
15 // store the value tostore in the matrix q,
16 // at row i, column j; m, n and
17 // s are as in padmalloc() above
18 void setter(float *q, int m, int n, int s,
19      int i, int j, float tostore) {
20   *(q + i*s+j) = tostore;
21 }
22
23 // fetch the value in the matrix q,
24 // at row i, column j; m, n and s are
25 // as in padmalloc() above
26 float getter(float *q, int m, int n, int s,
27      int i, int j) {
28   return *(q + i*s+j);
29 }
```

3.3 Interconnection Topologies

3.3.1 SMP Systems

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



Here and below:

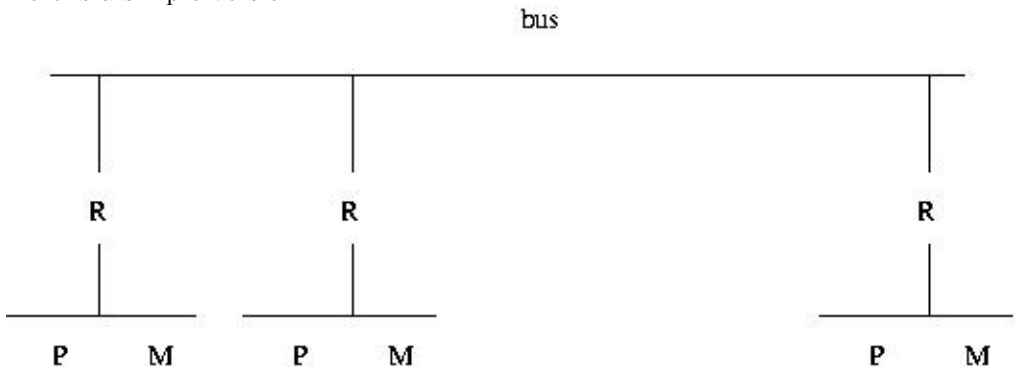
- The Ps are processors, e.g. off-the-shelf chips such as Pentiums.

- 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.

3.3.2 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.② 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.

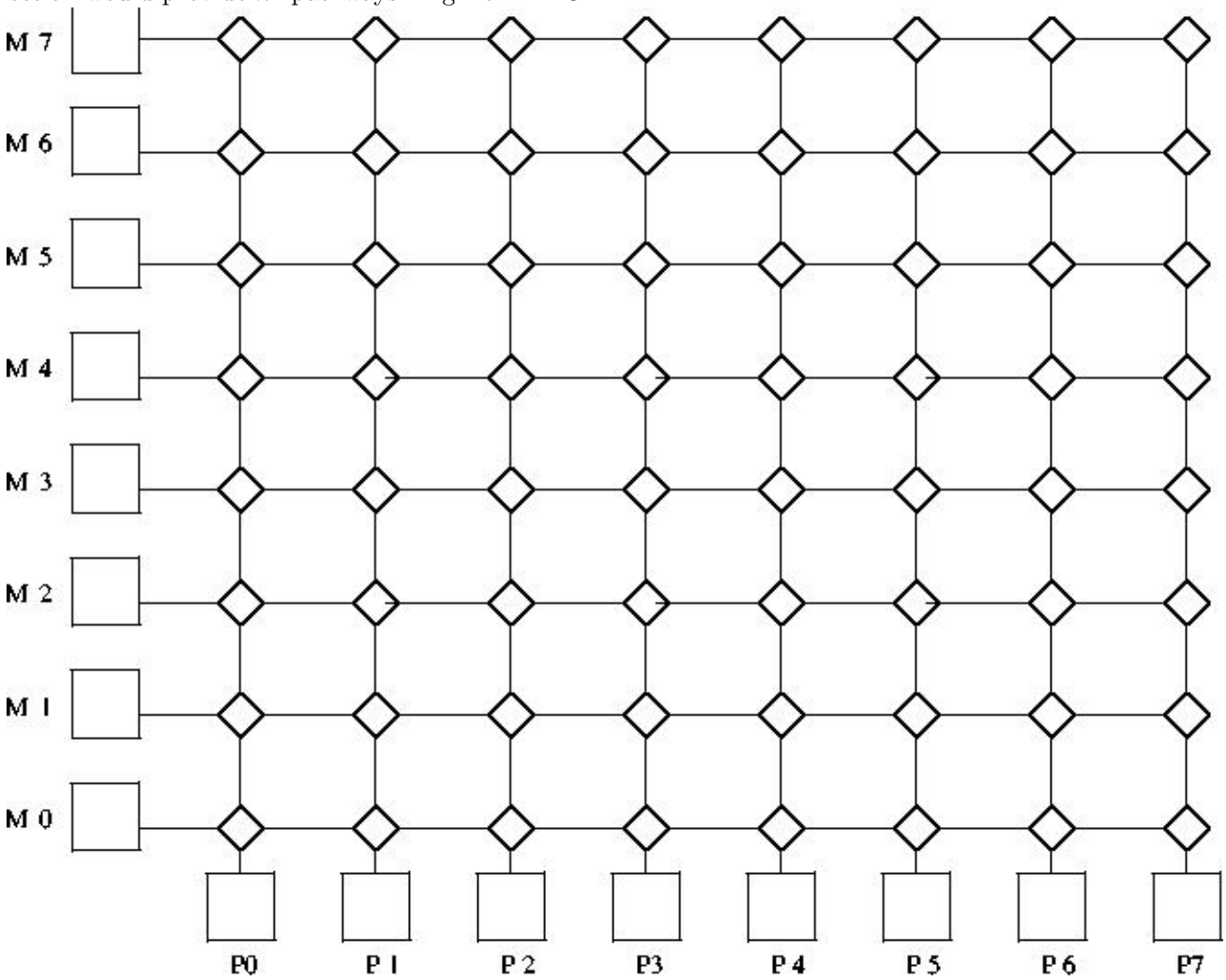
②This sounds similar to the concept of a cache. However, it is very different. A cache contains a local copy of some data stored elsewhere. Here it is the data itself, not a copy, which is being stored locally.

3.3.3 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.

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:



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.

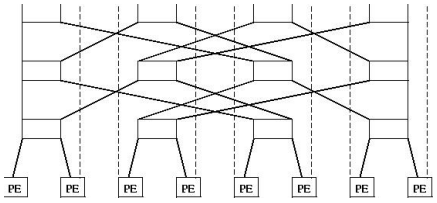
There could also be a return network of the same type, with this one being memory → processor, to return the result of the read requests.③

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.

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.④

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

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

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

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

③For safety's sake, i.e. fault tolerance, even writes are typically acknowledged in multiprocessor systems.

④The 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.

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

$$m = (2k + \lfloor (2k)/N \rfloor) \; mod \; N \tag{3.1}$$

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

3.3.4 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)	$O(\log_2 n)$	O(n)
bandwidth	O(1)	O(n)	O(n)
cost	O(1)	$O(n \log_2 n)$	$O(n^2)$

Let us see where these expressions come from, beginning with a bus: No matter how large n is, the time to get from, say, a processor to a memory module will be the same, thus O(1). Similarly, no matter how large n is, only one communication can occur at a time, thus again O(1).[5]

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

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

Similar considerations hold for the crossbar case.

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

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

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.

[5] 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.

3.3.5 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.

3.4 Synchronization Hardware

Avoidance of race conditions, e.g. implementation of locks, plays such a crucial role in shared-memory parallel processing that hardware assistance is a virtual necessity. Recall, for instance, that critical sections can effectively serialize a parallel program. Thus efficient implementation is crucial.

3.4.1 Test-and-Set 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.[6] 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:

```
1    copy L to R
2    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, using a register R, would be done by having the following code in the program being run:

```
1    TRY:  TAS R,L
2          JNZ TRY
```

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

```
    C:  ...   ; start of critical section
        ...
        ...   ; end of critical section
        MOV 0,L  ; 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.

LOCK Prefix on Intel Processors

On Pentium machines, the LOCK prefix can be used to get atomicity for certain instructions: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD. The bus will be locked for the duration of the execution of the instruction, thus setting up atomic operations. There is a special LOCK line in the control bus for this purpose. (Locking thus only applies to these instructions in forms in which there is an operand in memory.) By the way, XCHG asserts this LOCK# bus signal even if the LOCK prefix is not specified.

For example, consider our count-the-2s example on page **??**. If we store **mycount** in a register, say EDX, then

```
lock add %edx, overallcount
```

would replace the code we had earlier,

```
pthread_mutex_lock(&nextbaselock);
overallcount += mylocalcount;
pthread_mutex_unlock(&nextbaselock);
```

without locks!

Here is how we could implement a lock if needed. The lock would be in a variable named, say, **lockvar**.

```
  movl $lockvar, %ebx
  movl $1, %ecx
top:
  movl $0, %eax
  lock cmpxchg (%ebx), %ecx
  jz top # else leave the loop and enter the critical section
```

The operation CMPXCHG has EAX as an unnamed operand. The instruction basically does (here **source** is ECX and **destination** is **lockvar**)

```
if c(EAX) != c(destination) # sorry, lock is already locked
  c(EAX) <- c(destination)
  ZF <- 0 # the Zero Flag in the EFLAGS register
else
  c(destination) <- c(source) # lock the lock
  ZF <- 1
```

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 **overallcount**, and the second the write the new, incremented value back to **overallcount**. So, we are locking for a rather long time, potentially compromising performance when other threads want to access memory, but the benefits can be huge.

Locks with More Complex Interconnects

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.

3.4.2 May Not Need the Latest

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.

3.4.3 Fetch-and-Add Instructions

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

```
1 LOCK(K);
2 Y = X++;
3 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,

```
1 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.

3.5 Cache Issues

If you need a review of cache memories or don't have background in that area at all, read Section **??** in the appendix of this book before continuing.

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

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

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

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

- Under an invalidate protocol, when a processor writes to a variable in a cache, it first (i.e. before actually doing the write) tells each other cache to mark as invalid its cache line (if any)

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

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 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

```
1 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

[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.

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

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 positive feature in terms of cache coherency, because it makes broadcast very easy: Since everyone is attached to that single pathway, sending a message to all of them costs no more than sending it to just one—we get the others for free. That's no longer the case for multipath systems. In such systems, extra copies of the message must be created for each path, adding to overall traffic.

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

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

3.5.2 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:

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

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.

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	newstat
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

[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.

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.

3.5.3 The Problem of "False Sharing"

Consider the C declaration

```
1 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.

One possible solution is to add padding, e.g. declaring **W** and **Z** like this:

```
1 int W,U[1000],Z;
```

to separate **W** and **Z** so that they won't be in the same cache block. Of course, we must take block size into account, and check whether the compiler really has placed the two variables are in widely separated locations. To do this, we could for instance run the code

```
1 printf("%x %x\n,&W,&Z);
```

3.6 Memory-Access Consistency Policies

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

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

A related issue is that operations may occur, or appear to occur, out of order. As noted above, a read which follows a write in the program may execute before the write is sent to memory. Also, in

a multiprocessor system with multiple paths between processors and memory modules, two writes might take different paths, one longer than the other, and arrive "out of order." In order to simplify the presentation here, we will focus on the case in which the problem is due to write buffers, though.

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

The strongest consistency model is Sequential Consistency. It essentially requires that memory operations done by one processor are observed by the other processors to occur in the same order as executed on the first processor. Enforcement of this requirement makes a system slow, and it has been replaced on most systems by weaker models.

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

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

[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.

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.

3.7 Fetch-and-Add Combining within 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.

3.8 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 L1 cache. The chip may interface to the bus or interconnect network of via an L3 cache.

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

3.9 Optimal Number of Threads

A common question involves the best number of threads to run in a shared-memory setting. Clearly there is no general magic answer, but here are some considerations:[15]

- If your application does a lot of I/O, CPUs or cores may stay idle while waiting for I/O events. It thus makes sense to have many threads, so that computation threads can run when the I/O threads are tied up.
- In a purely computational application, one generally should not have more threads than cores. However, a program with a lot of virtual memory page faults may benefit from setting up extra threads, as page replacement involves (disk) I/O.
- Applications in which there is heavy interthread communication, say due to having a lot of lock variable, access, may benefit from setting up fewer threads than the number of cores.
- Many Intel processors include hardware for *hypertheading*. These are not full threads in the sense of having separate cores, but rather involve a limited amount of resource duplication within a core. The performance gain from this is typically quite modest. In any case, be aware of it; some software systems count these as threads, and assume for instance that there are 8 cores when the machine is actually just quad core.
- With GPUs (Chapter 5), most memory accesses have long latency and thus are I/O-like. Typically one needs very large numbers of threads for good performance.

3.10 Processor Affinity

With a timesharing OS, a given thread may run on different cores during different timeslices. If so, the cache for a given core may need a lot of refreshing, each time a new thread runs on that core. To avoid this slowdown, one might designate a preferred core for each thread, in the hope of reusing cache contents. Setting this up is dependent on the chip and the OS. OpenMP 3.1 has some facility for this.

3.11 Illusion of Shared-Memory through Software

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

[15] As with many aspects of parallel programming, a good basic knowledge of operating systems is key. See the reference on page 5.

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.[16] 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-based SDSMs work, consider the line of JIAJIA code

```
1 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

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

this action will eventually trigger a network transaction (not visible to the programmer) to the other JIAJIA nodes.[17] This transaction will then update the copies of **Prime** at the other nodes.[18]

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, you need a basic knowledge of how VM systems work. If you lack this, or need review, read Section **??** in the appendix of this book before continuing.

Here is how VM 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**.

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

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

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

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**:

```
1 int Y,*P;
2 ...
3 ...
4 P = &Y;
5 ...
```

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

All SDSM systems must deal with a software analog of the cache coherency problem. Whenever one node modifies the value of a shared variable, that node must notify the other nodes that a change has been made. The designer of the system must choose between update or invalidate protocols, just as in the hardware case.[19] 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.

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.

[19] 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.

Following is a JIAJIA example program, performing Odd/Even Transposition Sort. This is a variant on Bubble Sort, sometimes useful in parallel processing contexts.[20] 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
// 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
float *x;  // array to be sorted

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


// does sort of m-element array y
void oddeven(float *y, int m)
{  int i,left=jiapid-1,right=jiapid+1;
   float 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 = (float *) jia_alloc(n*sizeof(float));
   // barrier recommended after allocation
   jia_barrier();
```

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

```
55    // node 0 gets simple test array from command-line
56    if (jiapid == 0)  {
57       for (i = 0; i < n; i++)
58          x[i] = atoi(argv[i+3]);
59    }
60    jia_barrier();
61    if (debug && jiapid == 0)
62       while (mywait == 0)  { ; }
63    jia_barrier();
64    oddeven(x,n);
65    if (jiapid == 0)  {
66       printf("\nfinal array\n");
67       for (i = 0; i < n; i++)
68          printf("%f\n",x[i]);
69    }
70    jia_exit();
71 }
```

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.[21]

Each page—and thus each shared variable—has a **home** processor. If another processor writes to a page, then later when it reaches the unlock operation it must send all changes it made to the page back to the home node. In other words, the second processor calls **jia_unlock()**, which sends the changes to its sister invocation of **jia_unlock()** at the home processor.[22] 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,

[21] 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.

[22] 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.

listing where each home page resides.[23] 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.[24] When the barrier is reached, each will be informed of the writes of the other.[25] Allowing multiple writers helps to reduce the performance penalty due to false sharing.

3.12 Barrier Implementation

Recall that a **barrier** is program code[26] which has a processor do a wait-loop action until all processors have reached that point in the program.[27]

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.

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

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

[25] 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.

[26] Some hardware barriers have been proposed.

[27] 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.

3.12.1 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
       EvenOdd,  // "parity"
       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.

3.12.2 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;
   else 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**.

3.12.3 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]++;
   if (OldCount == PB->NNodes-1)  {
      PB->Count[Par] = 0;
      PB->EvenOdd = 1 - Par;
      pthread_mutex_unlock(&PB->Lock);
   }
   else {
      pthread_mutex_unlock(&PB->Lock);
      while (PB->Count[Par] > 0) ;
   }
}
```

3.12.4 Refinements

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->CV);
```

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

Parallelizing the Barrier Operation

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.

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.

第 4 章　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 Example: Dijkstra Shortest-Path Algorithm

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; could placed them
// above the "parallel" pragma in dowork()

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
```

```
22      mv,  // vertex which achieves that min
23      largeint = -1;  // max possible unsigned int
24
25  unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
26           // ohd[i*nv+j]
27           *mind;  // min distances found so far
28
29  void init(int ac, char **av)
30  {  int i,j,tmp;
31     nv = atoi(av[1]);
32     ohd = malloc(nv*nv*sizeof(int));
33     mind = malloc(nv*sizeof(int));
34     notdone = malloc(nv*sizeof(int));
35     // random graph
36     for (i = 0; i < nv; i++)
37        for (j = i; j < nv; j++)  {
38           if (j == i) ohd[i*nv+i] = 0;
39           else  {
40              ohd[nv*i+j] = rand() % 20;
41              ohd[nv*j+i] = ohd[nv*i+j];
42           }
43        }
44     for (i = 1; i < nv; i++)  {
45        notdone[i] = 1;
46        mind[i] = ohd[i];
47     }
48  }
49
50  // finds closest to 0 among notdone, among s through e
51  void findmymin(int s, int e, unsigned *d, int *v)
52  {  int i;
53     *d = largeint;
54     for (i = s; i <= e; i++)
55        if (notdone[i] && mind[i] < *d)  {
56           *d = mind[i];
57           *v = i;
58        }
59  }
60
61  // for each i in [s,e], ask whether a shorter path to i exists, through
62  // mv
63  void updatemind(int s, int e)
64  {  int i;
65     for (i = s; i <= e; i++)
66        if (mind[mv] + ohd[mv*nv+i] < mind[i])
67           mind[i] = mind[mv] + ohd[mv*nv+i];
68  }
69
70  void dowork()
71  {
72     #pragma omp parallel
73     {  int startv,endv,  // start, end vertices for my thread
74           step,  // whole procedure goes nv steps
75           mymv,  // vertex which attains the min value in my chunk
76           me = omp_get_thread_num();
77           unsigned mymd;  // min value found by this thread
78        #pragma omp single
79        {  nth = omp_get_num_threads();  // must call inside parallel block
80           if (nv % nth != 0) {
81              printf("nv must be divisible by nth\n");
```

```
82             exit(1);
83          }
84          chunk = nv/nth;
85          printf("there are %d threads\n",nth);
86       }
87       startv = me * chunk;
88       endv = startv + chunk - 1;
89       for (step = 0; step < nv; step++)  {
90          // find closest vertex to 0 among notdone; each thread finds
91          // closest in its group, then we find overall closest
92          #pragma omp single
93          {  md = largeint; mv = 0;  }
94          findmymin(startv,endv,&mymd,&mymv);
95          // update overall min if mine is smaller
96          #pragma omp critical
97          {  if (mymd < md)
98                {  md = mymd; mv = mymv;  }
99          }
100         #pragma omp barrier
101         // mark new vertex as done
102         #pragma omp single
103         {  notdone[mv] = 0;  }
104         // now update my section of mind
105         updatemind(startv,endv);
106         #pragma omp barrier
107      }
108   }
109}
110
111int main(int argc, char **argv)
112{  int i,j,print;
113   double startime,endtime;
114   init(argc,argv);
115   startime = omp_get_wtime();
116   // parallel
117   dowork();
118   // back to single thread
119   endtime = omp_get_wtime();
120   printf("elapsed time:  %f\n",endtime-startime);
121   print = atoi(argv[2]);
122   if (print)  {
123      printf("graph weights:\n");
124      for (i = 0; i < nv; i++)  {
125         for (j = 0; j < nv; j++)
126            printf("%u  ",ohd[nv*i+j]);
127         printf("\n");
128      }
129      printf("minimum distances:\n");
130      for (i = 1; i < nv; i++)
131         printf("%u\n",mind[i]);
132   }
133}
```

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 between 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;
```

[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.

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

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

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

4.2.3 Scope Issues

Note carefully that in

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

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 **private** keyword and its cousins. For instance,

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

would make **x** and **y** nonshared even if they were declared above the directive line. You may wish to modify that a bit, so that **x** and **y** have initial values that were shared before the directive; use **firstprivate** for this.

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.:

```
1      #pragma omp single
2      {  nth = omp_get_num_threads();
3         if (nv % nth != 0) {
4            printf("nv must be divisible by nth\n");
5            exit(1);
6         }
7         chunk = nv/nth;
8         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**.

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

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.

```
#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. (The threads, of course, must have already been set up via the **omp parallel** pragma.) 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 Example: Dijkstra with Parallel for Loops

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>

```

```
14 // global variables, shared by all threads by default
15
16 int nv,  // number of vertices
17     *notdone, // vertices not checked yet
18     nth,  // number of threads
19     chunk,  // number of vertices handled by each thread
20     md,  // current min over all threads
21     mv,  // vertex which achieves that min
22     largeint = -1;  // max possible unsigned int
23
24 unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
25                 // ohd[i*nv+j]
26          *mind;  // min distances found so far
27
28 void init(int ac, char **av)
29 {  int i,j,tmp;
30    nv = atoi(av[1]);
31    ohd = malloc(nv*nv*sizeof(int));
32    mind = malloc(nv*sizeof(int));
33    notdone = malloc(nv*sizeof(int));
34    // random graph
35    for (i = 0; i < nv; i++)
36       for (j = i; j < nv; j++)  {
37          if (j == i) ohd[i*nv+i] = 0;
38          else  {
39             ohd[nv*i+j] = rand() % 20;
40             ohd[nv*j+i] = ohd[nv*i+j];
41          }
42       }
43    for (i = 1; i < nv; i++)  {
44       notdone[i] = 1;
45       mind[i] = ohd[i];
46    }
47 }
48
49 void dowork()
50 {
51    #pragma omp parallel
52    {  int step,  // whole procedure goes nv steps
53           mymv,  // vertex which attains that value
54           me = omp_get_thread_num(),
55           i;
56       unsigned mymd;  // min value found by this thread
57       #pragma omp single
58       {  nth = omp_get_num_threads();
59          printf("there are %d threads\n",nth);  }
60       for (step = 0; step < nv; step++)  {
61          // find closest vertex to 0 among notdone; each thread finds
62          // closest in its group, then we find overall closest
63          #pragma omp single
64          {  md = largeint; mv = 0;  }
65          mymd = largeint;
66          #pragma omp for nowait
67          for (i = 1; i < nv; i++)  {
68             if (notdone[i] && mind[i] < mymd)  {
69                mymd = ohd[i];
70                mymv = i;
71             }
72          }
73          // update overall min if mine is smaller
```

```
74          #pragma omp critical
75          {  if (mymd < md)
76               {  md = mymd; mv = mymv;  }
77          }
78          // mark new vertex as done
79          #pragma omp single
80          {  notdone[mv] = 0;  }
81          // now update ohd
82          #pragma omp for
83          for (i = 1; i < nv; i++)
84             if (mind[mv] + ohd[mv*nv+i] < mind[i])
85                mind[i] = mind[mv] + ohd[mv*nv+i];
86       }
87    }
88 }
89
90 int main(int argc, char **argv)
91 {  int i,j,print;
92    init(argc,argv);
93    // parallel
94    dowork();
95    // back to single thread
96    print = atoi(argv[2]);
97    if (print)  {
98       printf("graph weights:\n");
99       for (i = 0; i < nv; i++)  {
100         for (j = 0; j < nv; j++)
101            printf("%u  ",ohd[nv*i+j]);
102         printf("\n");
103      }
104      printf("minimum distances:\n");
105      for (i = 1; i < nv; i++)
106         printf("%u\n",mind[i]);
107   }
108 }
109
```

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

```
1           #pragma omp for
2           for (i = 1; i < nv; i++)  {
3              if (notdone[i] && mind[i] < mymd)  {
4                 mymd = ohd[i];
5                 mymv = i;
6              }
7           }
```

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: the schedule Clause

In this default version of the **for** construct, iterations are executed by threads *in unpredictable order*; the OpenMP standard does not specify which threads will execute which iterations in which order. But this can be controlled by the programmer, using the **schedule** clause. OpenMP provides three choices for this:

- **static:** The iterations are grouped into chunks, and assigned to threads in round-robin fashion. Default chunk size is approximately the number of iterations divided by the number of threads.
- **dynamic:** Again the iterations are grouped into chunks, but here the assignment of chunks to threads is done dynamically. When a thread finishes working on a chunk, it asks the OpenMP runtime system to assign it the next chunk in the queue. Default chunk size is 1.
- **guided:** Similar to dynamic, but with the chunk size decreasing as execution proceeds.

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 get the same operation with less code by asking OpenMP to do the chunking for us, say with a chunk size of 8:

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

Note again that this would have the same effect as our original code, which each thread handling one chunk of contiguous iterations within a loop. So it's just a programming convenience for us in this case. (If the number of threads doesn't evenly divide the number of iterations, OpenMP will fix that up for us too.)

The more general form is

```
#pragma omp for schedule(static,chunk)
```

Here **static** is still a keyword but **chunk** is an actual argument. However, setting the chunk size in the **schedule()** clause is a *compile-time* operation. If you wish to have the chunk size set at run time, call **omp_set_schedule()** in conjunction with the **runtime** clause. Example:

```
int main(int argc, char **argv)
{
   ...
   n = atoi(argv[1]);
   int chunk = atoi(argv[2]);
   omp_set_schedule(omp_sched_static, chunk);
   #pragma omp parallel
   {
      ...
      #pragma omp for schedule(runtime)
      for (i = 1; i < n; i++) {
         ...
      }
      ...
   }
}
```

Or set the **OMP_SCHEDULE** environment variable.

The syntax is the same for **dynamic** and **guided**.

As discussed in Section **??**, 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];
...
```

There are other variations of this available in OpenMP. However, in Section **??**, I showed that these would seldom be necessary or desirable; having each thread handle a single chunk would be best.

See Section **??** for a timing example.

```
setenv OMP_SCHEDULE "static,20"
```

4.3.4 Example: In-Place Matrix Transpose

This method works in-place, a virtue if we are short on memory. Its cache performance is probably poor, though. It may be better to look at horizontal slabs above the diagonal, say, and trade them with vertical ones below the diagonal.

```
1  #include <omp.h>
2
3  // translate from 2-D to 1-D indices
4  int onedim(int n,int i,int j) { return n * i + j; }
5
6  void transp(int *m, int n)
7  {
8     #pragma omp parallel
9     { int i,j,tmp;
10       // walk through all the above-diagonal elements, swapping them
11       // with their below-diagonal counterparts
12       #pragma omp for
13       for (i = 0; i < n; i++) {
14          for (j = i+1; j < n; j++) {
15             tmp = m[onedim(n,i,j)];
16             m[onedim(n,i,j)] = m[onedim(n,j,i)];
17             m[onedim(n,j,i)] = tmp;
18          }
19       }
20    }
21 }
```

4.3.5 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

```
1 int z;
2 ...
3 #pragma omp for reduction(+:z)
4 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.

One can specify several reduction variables to the right of the colon, separated by commas.

Our use of the **reduction** clause here makes our programming much easier. Indeed, if we 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
^	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]

Note that the reduction variables must be shared by the threads, and apparently the only acceptable way to do so in this case is to declare them as global variables.

A reduction variable must be scalar, in C/C++. It can be an array in FORTRAN.

4.4 Example: Mandelbrot Set

Here's the code for the timings in Section **??**:

```
// compile with -D, e.g.
//
// gcc -fopenmp -o manddyn Gove.c -DDYNAMIC
```

[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.

```
4  //
5  // to get the version that uses dynamic scheduling
6
7  #include <omp.h>
8  #include <complex.h>
9
10 #include <time.h>
11 float timediff(struct timespec t1, struct timespec t2)
12 { if (t1.tv_nsec > t2.tv_nsec) {
13         t2.tv_sec -= 1;
14         t2.tv_nsec += 1000000000;
15    }
16    return t2.tv_sec-t1.tv_sec + 0.000000001 * (t2.tv_nsec-t1.tv_nsec);
17 }
18
19 #ifdef RC
20 // finds chunk among 0,...,n-1 to assign to thread number me among nth
21 // threads
22 void findmyrange(int n,int nth,int me,int *myrange)
23 { int chunksize = n / nth;
24    myrange[0] = me * chunksize;
25    if (me < nth-1) myrange[1] = (me+1) * chunksize - 1;
26    else myrange[1] = n - 1;
27 }
28
29 #include <stdlib.h>
30 #include <stdio.h>
31 // from http://www.cis.temple.edu/~ingargio/cis71/code/randompermute.c
32 // It returns a random permutation of 0..n-1
33 int * rpermute(int n) {
34   int *a = (int *)(int *) malloc(n*sizeof(int));
35   // int *a = malloc(n*sizeof(int));
36   int k;
37   for (k = 0; k < n; k++)
38     a[k] = k;
39   for (k = n-1; k > 0; k--) {
40     int j = rand() % (k+1);
41     int temp = a[j];
42     a[j] = a[k];
43     a[k] = temp;
44   }
45   return a;
46 }
```

```
47 #endif
48
49 #define MAXITERS 1000
50
51 // globals
52 int count = 0;
53 int nptsside;
54 float side2;
55 float side4;
56
57 int inset(double complex c) {
58    int iters;
59    float rl,im;
60    double complex z = c;
61    for (iters = 0; iters < MAXITERS; iters++) {
62       z = z*z + c;
63       rl = creal(z);
64       im = cimag(z);
65       if (rl*rl + im*im > 4) return 0;
66    }
67    return 1;
68 }
69
70 int *scram;
71
72 void dowork()
73 {
74    #ifdef RC
75    #pragma omp parallel reduction(+:count)
76    #else
77    #pragma omp parallel
78    #endif
79    {
80       int x,y; float xv,yv;
81       double complex z;
82       #ifdef STATIC
83       #pragma omp for reduction(+:count) schedule(static)
84       #elif defined DYNAMIC
85       #pragma omp for reduction(+:count) schedule(dynamic)
86       #elif defined GUIDED
87       #pragma omp for reduction(+:count) schedule(guided)
88       #endif
89       #ifdef RC
```

```
90       int myrange[2];
91       int me = omp_get_thread_num();
92       int nth = omp_get_num_threads();
93       int i;
94       findmyrange(nptsside,nth,me,myrange);
95       for (i = myrange[0]; i <= myrange[1]; i++) {
96          x = scram[i];
97       #else
98       for (x=0; x<nptsside; x++) {
99       #endif
100         for ( y=0; y<nptsside; y++) {
101            xv = (x - side2) / side4;
102            yv = (y - side2) / side4;
103            z = xv + yv*I;
104            if (inset(z)) {
105               count++;
106            }
107         }
108      }
109   }
110 }
111
112 int main(int argc, char **argv)
113 {
114   nptsside = atoi(argv[1]);
115   side2 = nptsside / 2.0;
116   side4 = nptsside / 4.0;
117
118   struct timespec bgn,nd;
119   clock_gettime(CLOCK_REALTIME, &bgn);
120
121   #ifdef RC
122   scram = rpermute(nptsside);
123   #endif
124
125   dowork();
126
127   // implied barrier
128   printf("%d\n",count);
129   clock_gettime(CLOCK_REALTIME, &nd);
130   printf("%f\n",timediff(bgn,nd));
131 }
```

The code is similar to that of a number of books and Web sites, such as the Gove book cited in Section **??**. Here RC is the random chunk method discussed in Section **??**.

4.5 The Task Directive

This is new to OpenMP 3.0. The basic idea is to set up a task queue: When a thread encounters a **task** directive, it arranges for some thread to execute the associated block—at some time. 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. Also, there may be more tasks than threads, also causing some threads to wait.

Note that we could arrange for all this ourselves, without **task**. We'd set up our own work queue, as a shared variable, and write our code so that whenever a thread finished a unit of work, it would delete the head of the queue. Whenever a thread generated a unit of work, it would add it to the que. Of course, the deletion and addition would have to be done atomically. All this would amount to a lot of coding on our part, so **task** really simplifies the programming.

4.5.1 Example: Quicksort

```
1  // OpenMP example program:  quicksort; not necessarily efficient
2
3  void swap(int *yi, int *yj)
4  {  int tmp = *yi;
5     *yi = *yj;
6     *yj = tmp;
7  }
8
9  int *separate(int *x, int low, int high)
10 {  int i,pivot,last;
11    pivot = x[low];  // would be better to take, e.g., median of 1st 3 elts
12    swap(x+low,x+high);
13    last = low;
14    for (i = low; i < high; i++) {
15       if (x[i] <= pivot) {
16          swap(x+last,x+i);
17          last += 1;
18       }
19    }
20    swap(x+last,x+high);
21    return last;
22 }
23
24 // quicksort of the array z, elements zstart through zend; set the
25 // latter to 0 and m-1 in first call, where m is the length of z;
26 // firstcall is 1 or 0, according to whether this is the first of the
27 // recursive calls
28 void qs(int *z, int zstart, int zend, int firstcall)
29 {
30    #pragma omp parallel
31    {  int part;
32       if (firstcall == 1) {
33          #pragma omp single nowait
34          qs(z,0,zend,0);
35       } else {
```

```
36          if (zstart < zend) {
37             part = separate(z,zstart,zend);
38             #pragma omp task
39             qs(z,zstart,part-1,0);
40             #pragma omp task
41             qs(z,part+1,zend,0);
42          }
43
44       }
45    }
46 }
47
48 // test code
49 main(int argc, char**argv)
50 {  int i,n,*w;
51    n = atoi(argv[1]);
52    w = malloc(n*sizeof(int));
53    for (i = 0; i < n; i++) w[i] = rand();
54    qs(w,0,n-1,1);
55    if (n < 25)
56       for (i = 0; i < n; i++) printf("%d\n",w[i]);
57 }
```

The code

```
1 if (firstcall == 1) {
2    #pragma omp single nowait
3    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

```
1 part = separate(z,zstart,zend);
2 #pragma omp task
3 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."

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

4.6 Other OpenMP Synchronization Issues

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

4.6.1 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.

```
1 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.

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

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

The eligible operators are:

```
1 ++, --, +=, *=, <<=, &=, |=
```

4.6.2 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 as discussed in Section 3.6, 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:

```
1 #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.

4.7 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:

```
1 #pragma omp parallel for
```

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

4.8 The Rest of OpenMP

There is much, much more to OpenMP than what we have seen here. To see the details, there are many Web pages you can check, and there is also the excellent book, *Using OpenMP: Portable Shared Memory Parallel Programming*, by Barbara Chapman, Gabriele Jost and Ruud Van Der Pas, MIT Press, 2008. The book by Gove cited in Section **??** also includes coverage of OpenMP.

4.9 Compiling, Running and Debugging OpenMP Code

4.9.1 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

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

  Note: Apparently declarations of local variables cannot be made in the midst of code; they must precede all code within a block.

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

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

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

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

  You can also use **-lgomp** instead of **-fopenmp**.

4.9.2 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

```
1 setenv OMP_NUM_THREADS 4
```

4.9.3 Debugging

Since OpenMP is essentially just an interface to threads, your debugging tool's threads facilities should serve you well. See Section 1.3.2 for the GDB case.

A possible problem, though, is that 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

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

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:

- 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 was fixed in version 4.4 of the GCC suite, but seems to have slipped back in with some later versions! This may be due to compiler optimizations that place variables in registers.
- 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: Older versions also changed your function names, but the current version (1.2.0) doesn't. Works fine in GDB.

4.10 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.

4.10.1 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

4.10.2 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:

```
1  // Dijkstra.c
2
3  // OpenMP example program:  Dijkstra shortest-path finder in a
4  // bidirectional graph; finds the shortest path from vertex 0 to all
5  // others
6
7  // **** in this version, instead of having a critical section in which
8  // each thread updates md and mv, the threads record their mymd and mymv
9  // values in a global array mymins, which one thread then later uses to
10 // update md and mv
11
12 // usage:  dijkstra nv print
13
14 // where nv is the size of the graph, and print is 1 if graph and min
15 // distances are to be printed out, 0 otherwise
16
17 #include <omp.h>
18
19 // global variables, shared by all threads by default
20
21 int nv,  // number of vertices
22     *notdone, // vertices not checked yet
23     nth,  // number of threads
24     chunk,  // number of vertices handled by each thread
25     md,  // current min over all threads
26     mv,  // vertex which achieves that min
27     largeint = -1;  // max possible unsigned int
28
29 int *mymins;  // (mymd,mymv) for each thread; see dowork()
30
31 unsigned *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
32          // ohd[i*nv+j]
33          *mind;  // min distances found so far
34
35 void init(int ac, char **av)
36 {  int i,j,tmp;
37    nv = atoi(av[1]);
38    ohd = malloc(nv*nv*sizeof(int));
39    mind = malloc(nv*sizeof(int));
40    notdone = malloc(nv*sizeof(int));
41    // random graph
42    for (i = 0; i < nv; i++)
43       for (j = i; j < nv; j++)  {
44          if (j == i) ohd[i*nv+i] = 0;
45          else  {
46             ohd[nv*i+j] = rand() % 20;
47             ohd[nv*j+i] = ohd[nv*i+j];
48          }
49       }
50    for (i = 1; i < nv; i++)  {
51       notdone[i] = 1;
```

```
52          mind[i] = ohd[i];
53       }
54    }
55
56    // finds closest to 0 among notdone, among s through e
57    void findmymin(int s, int e, unsigned *d, int *v)
58    {  int i;
59       *d = largeint;
60       for (i = s; i <= e; i++)
61          if (notdone[i] && mind[i] < *d)  {
62             *d = ohd[i];
63             *v = i;
64          }
65    }
66
67    // for each i in [s,e], ask whether a shorter path to i exists, through
68    // mv
69    void updatemind(int s, int e)
70    {  int i;
71       for (i = s; i <= e; i++)
72          if (mind[mv] + ohd[mv*nv+i] < mind[i])
73             mind[i] = mind[mv] + ohd[mv*nv+i];
74    }
75
76    void dowork()
77    {
78       #pragma omp parallel
79       {  int startv,endv,  // start, end vertices for my thread
80             step,  // whole procedure goes nv steps
81             me,
82             mymv;  // vertex which attains the min value in my chunk
83             unsigned mymd;  // min value found by this thread
84          int i;
85          me = omp_get_thread_num();
86          #pragma omp single
87          {  nth = omp_get_num_threads();
88             if (nv % nth != 0) {
89                printf("nv must be divisible by nth\n");
90                exit(1);
91             }
92             chunk = nv/nth;
93             mymins = malloc(2*nth*sizeof(int));
94          }
95          startv = me * chunk;
96          endv = startv + chunk - 1;
97          for (step = 0; step < nv; step++)  {
98             // find closest vertex to 0 among notdone; each thread finds
99             // closest in its group, then we find overall closest
100            findmymin(startv,endv,&mymd,&mymv);
101            mymins[2*me] = mymd;
102            mymins[2*me+1] = mymv;
103            #pragma omp barrier
104            // mark new vertex as done
105            #pragma omp single
106            {  md = largeint; mv = 0;
107               for (i = 1; i < nth; i++)
108                  if (mymins[2*i] < md) {
109                     md = mymins[2*i];
110                     mv = mymins[2*i+1];
111                  }
```

```
112             notdone[mv] = 0;
113          }
114          // now update my section of mind
115          updatemind(startv,endv);
116          #pragma omp barrier
117       }
118    }
119 }
120
121 int main(int argc, char **argv)
122 {  int i,j,print;
123    double startime,endtime;
124    init(argc,argv);
125    startime = omp_get_wtime();
126    // parallel
127    dowork();
128    // back to single thread
129    endtime = omp_get_wtime();
130    printf("elapsed time:  %f\n",endtime-startime);
131    print = atoi(argv[2]);
132    if (print)  {
133       printf("graph weights:\n");
134       for (i = 0; i < nv; i++)  {
135          for (j = 0; j < nv; j++)
136             printf("%u  ",ohd[nv*i+j]);
137          printf("\n");
138       }
139       printf("minimum distances:\n");
140       for (i = 1; i < nv; i++)
141          printf("%u\n",mind[i]);
142    }
143 }
```

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

```
1          findmymin(startv,endv,&mymd,&mymv);
2          mymins[2*me] = mymd;
3          mymins[2*me+1] = mymv;
4          #pragma omp barrier
5          // mark new vertex as done
6          #pragma omp single
7          {  notdone[mv] = 0;
8             for (i = 1; i < nth; i++)
9                if (mymins[2*i] < md) {
10                  md = mymins[2*i];
11                  mv = mymins[2*i+1];
12               }
13         }
14         // now update my section of mind
15         updatemind(startv,endv);
16         #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.

4.10.3 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:

```
1  # 93 "Dijkstra.c"
2  findmymin(startv,endv,&(mymd),&(mymv));{
3  _ompc_enter_critical(&__ompc_lock_critical);
4  # 96 "Dijkstra.c"
5  if((mymd)<(((unsigned )(md)))){
6
7  # 97 "Dijkstra.c"
8  (md)=(((int )(mymd)));
9  # 97 "Dijkstra.c"
10 (mv)=(mymv);
11 }_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.

With Ompi, while preprocessing of your file **x.c**, the compiler produces an intermediate 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! So it's a bit hard to read, but can be very instructive.

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.

4.11 Example: Root Finding

The application is described in the comments, but here are a couple of things to look for in particular:

- The variables **curra** and **currb** are shared by all the threads, but due to the nature of the application, no critical sections are needed.
- On the other hand, the barrier is essential. The reader should ponder what calamities would occur without it.

Note the disclaimer in the comments, to the effect that parallelizing this application will be fruitful only if the functioin f() is very time-consuming to evaluate. It might be the output of some complex simulation, for instance, with the argument to f() being some simulation parameter.

```
1  #include<omp.h>
2  #include<math.h>
3
4  // OpenMP example: root finding
5
6  // the function f() is known to be negative
7  // at a, positive at b, and thus has at
8  // least one root in (a,b); if there are
9  // multiple roots, only one is found;
10 // the procedure runs for niters iterations
11
12 // strategy: in each iteration, the current
13 // interval is split into nth equal parts,
14 // and each thread checks its subinterval
15 // for a sign change of f(); if one is
16 // found, this subinterval becomes the
17 // new current interval; the current guess
18 // for the root is the left endpoint of the
19 // current interval
20
21 // of course, this approach is useful in
22 // parallel only if f() is very expensive
23 // to evaluate
24
25 // for simplicity, assumes that no endpoint
26 // of a subinterval will ever exactly
27 // coincide with a root
28
29 float root(float(*f)(float),
30   float inita, float initb, int niters) {
31   float curra = inita;
32   float currb = initb;
33   #pragma omp parallel
34   {
35     int nth = omp_get_num_threads();
```

```
      int me = omp_get_thread_num();
      int iter;
      for (iter = 0; iter < niters; iter++) {
         #pragma omp barrier
         float subintwidth =
            (currb - curra) / nth;
         float myleft =
            curra + me * subintwidth;
         float myright = myleft + subintwidth;
         if ((*f)(myleft) < 0 &&
             (*f)(myright) > 0) {
            curra = myleft;
            currb = myright;
         }
      }
   }
   return curra;
}

float testf(float x) {
   return pow(x-2.1,3);
}

int main(int argc, char **argv)
{ printf("%f\n",root(testf,-4.1,4.1,1000)); }
```

4.12 Example: Mutual Outlinks

Consider the example of Section 2.4.3. We have 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 OpenMP code below finds the mean number of mutual outlinks, among all pairs of sites in a set of Web sites. Note that it uses the method for load balancing presented in Section 2.4.3.

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

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

int n,  // number of sites (will assume n is even)
    nth,  // number of threads (will assume n/2 divisible by nth)
    *m, // link matrix
    tot = 0; // grand total of matches

// processes row pairs (i,i+1), (i,i+2), ...
int procpairs(int i)
```

```
{  int j,k,sum=0;
   for (j = i+1; j < n; j++) {
      for (k = 0; k < n; k++)
         sum += m[n*i+k] * m[n*j+k];
   }
   return sum;
}

float dowork()
{
   #pragma omp parallel
   {  int pn1,pn2,i,mysum=0;
      int me = omp_get_thread_num();
      nth = omp_get_num_threads();
      // in checking all (i,j) pairs, partition the work according to i;
      // to get good load balance, this thread me will handle all i that equal
      // me mod nth
      for (i = me; i < n; i += nth) {
         mysum += procpairs(i);
      }
      #pragma omp atomic
      tot += mysum;
      #pragma omp barrier
   }
   int divisor = n * (n-1) / 2;
   return ((float) tot)/divisor;
}

int main(int argc, char **argv)
{   int n2 = n/2,i,j;
    n = atoi(argv[1]);  // number of matrix rows/cols
    int msize = n * n * sizeof(int);
    m = (int *) malloc(msize);
    // as a test, fill matrix with random 1s and 0s
    for (i = 0; i < n; i++) {
       m[n*i+i] = 0;
       for (j = 0; j < n; j++) {
          if (j != i) m[i*n+j] = rand() % 2;
       }
    }
    if (n < 10) {
       for (i = 0; i < n; i++) {
          for (j = 0; j < n; j++) printf("%d  ",m[n*i+j]);
          printf("\n");
       }
    }
    tot = 0;
    float meanml = dowork();
    printf("mean = %f\n",meanml);
}
```

4.13 Example: Transforming an Adjacency Matrix

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{4.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{4.2}$$

For instance, there is a 1 on the far right, second row of the above matrix, meaning that in the graph there is an edge from vertex 1 to vertex 3. This results in the row (1,3) in the transformed matrix seen above.

Suppose further that we require this listing to be in lexicographical order, sorted on source vertex and then on destination vertex. Here is code to do this computation in OpenMP:

```
1  // takes a graph adjacency matrix for a directed graph, and converts it
2  // to a 2-column matrix of pairs (i,j), meaning an edge from vertex i to
3  // vertex j; the output matrix must be in lexicographical order
4
5  // not claimed efficient, either in speed or in memory usage
6
7  #include <omp.h>
8
9  // needs -lrt link flag for C++
10 #include <time.h>
11 float timediff(struct timespec t1, struct timespec t2)
12 { if (t1.tv_nsec > t2.tv_nsec) {
13         t2.tv_sec -= 1;
14         t2.tv_nsec += 1000000000;
15    }
16    return t2.tv_sec-t1.tv_sec + 0.000000001 * (t2.tv_nsec-t1.tv_nsec);
17 }
18
19 // transgraph() does this work
20 // arguments:
21 // adjm: the adjacency matrix (NOT assumed symmetric), 1 for edge, 0
```

```
22 // otherwise; note: matrix is overwritten by the function
23 // n: number of rows and columns of adjm
24 // nout: output, number of rows in returned matrix
25 // return value: pointer to the converted matrix
26 int *transgraph(int *adjm, int n, int *nout)
27 {
28    int *outm, // to become the output matrix
29        *num1s, // i-th element will be the number of 1s in row i of adjm
30        *cumul1s; // cumulative sums in num1s
31    #pragma omp parallel
32    { int i,j,m;
33       int me = omp_get_thread_num(),
34           nth = omp_get_num_threads();
35       int myrows[2];
36       int tot1s;
37       int outrow,num1si;
38       #pragma omp single
39       {
40          num1s = malloc(n*sizeof(int));
41          cumul1s = malloc((n+1)*sizeof(int));
42       }
43       // determine the rows in adjm to be handled by this thread
44       findmyrange(n,nth,me,myrows);
45       // start the action
46       for (i = myrows[0]; i <= myrows[1]; i++) {
47          tot1s = 0;
48          for (j = 0; j < n; j++)
49             if (adjm[n*i+j] == 1) {
50                adjm[n*i+(tot1s++)] = j;
51             }
52          num1s[i] = tot1s;
53       }
54       #pragma omp barrier
55       #pragma omp single
56       {
57          cumul1s[0] = 0;
58          // now calculate where the output of each row in adjm
59          // should start in outm
60          for (m = 1; m <= n; m++) {
61             cumul1s[m] = cumul1s[m-1] + num1s[m-1];
62          }
63          *nout = cumul1s[n];
64          outm = malloc(2*(*nout) * sizeof(int));
```

```
65       }
66       // now fill in this thread's portion
67       for (i = myrows[0]; i <= myrows[1]; i++) {
68          outrow = cumul1s[i];
69          num1si = num1s[i];
70          for (j = 0; j < num1si; j++) {
71             outm[2*(outrow+j)] = i;
72             outm[2*(outrow+j)+1] = adjm[n*i+j];
73          }
74       }
75       #pragma omp barrier
76    }
77    return outm;
78 }
79 
80 int main(int argc, char **argv)
81 { int i,j;
82    int *adjm;
83    int n = atoi(argv[1]);
84    int nout;
85    int *outm;
86    adjm = malloc(n*n*sizeof(int));
87    for (i = 0; i < n; i++)
88       for (j = 0; j < n; j++)
89          if (i == j) adjm[n*i+j] = 0;
90          else adjm[n*i+j] = rand() % 2;
91 
92    struct timespec bgn,nd;
93    clock_gettime(CLOCK_REALTIME, &bgn);
94 
95    outm = transgraph(adjm,n,&nout);
96    printf("number of output rows: %d\n",nout);
97 
98    clock_gettime(CLOCK_REALTIME, &nd);
99    printf("%f\n",timediff(bgn,nd));
100 
101    if (n <= 10)
102       for (i = 0; i < nout; i++)
103          printf("%d %d\n",outm[2*i],outm[2*i+1]);
104 }
105 
106 // finds chunk among 0,...,n-1 to assign to thread number me among nth
107 // threads
```

```
108 void findmyrange(int n,int nth,int me,int *myrange)
109 { int chunksize = n / nth;
110   myrange[0] = me * chunksize;
111   if (me < nth-1) myrange[1] = (me+1) * chunksize - 1;
112   else myrange[1] = n - 1;
113 }
```

4.14 Locks with OpenMP

Though one of OpenMP's best virtues is that you can avoid working with those pesky lock variables needed for straight threads programming, there are still some instances in which lock variables may be useful. OpenMP does provide for locks:

- declare your locks to be of type `omp_lock_t`
- call `omp_set_lock()` to lock the lock
- call `omp_unset_lock()` to unlock the lock

4.15 Other Examples of OpenMP Code in This Book

There are additional OpenMP examples in later sections of this book, such as:[5]

- sampling bucket sort, Section 1.3.2
- parallel prefix sum/run-length decoding, Section 11.3.
- matrix multiplication, Section 12.3.2.
- Jacobi algorithm for solving systems of linear equations, with a good example of the OpenMP **reduction** clause, Section 12.5.4
- another implementation of Quicksort, Section 13.1.2

[5] If you are reading this presentation on OpenMP separately from the book, the book is at `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`

第 5 章　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 or games.

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, i.e. ways to disguising one's problem so that it appeared to be doing graphics computations. Though some high-level interfaces were developed to automate this transformation, effective coding required some understanding of graphics principles.

But current-generation 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. OpenCL is not yet in as broad use as CUDA, so our discussion here focuses on CUDA and NVIDIA GPUs.

Also, the discussion will focus on NVIDIA's Tesla line. This then led to the second generation, Fermi, and then Kepler. Unless otherwise stated, all statements here refer to Tesla.

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 Example: Calculate Row Sums

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  // CUDA example: finds row sums of an integer matrix m
6
7  // find1elt() finds the rowsum of one row of the nxn matrix m, storing the
8  // result in the corresponding position in the rowsum array rs; matrix
9  // stored as 1-dimensional, row-major order
10
11 __global__ void find1elt(int *m, int *rs, int n)
12 {
13    int rownum = blockIdx.x; // this thread will handle row # rownum
14    int sum = 0;
15    for (int k = 0; k < n; k++)
16       sum += m[rownum*n+k];
17    rs[rownum] = sum;
18 }
19
20 int main(int argc, char **argv)
21 {
22     int n = atoi(argv[1]); // number of matrix rows/cols
23     int *hm, // host matrix
24         *dm, // device matrix
25         *hrs, // host rowsums
26         *drs; // device rowsums
27     int msize = n * n * sizeof(int); // size of matrix in bytes
28     // allocate space for host matrix
29     hm = (int *) malloc(msize);
30     // as a test, fill matrix with consecutive integers
31     int t = 0,i,j;
32     for (i = 0; i < n; i++) {
```

```
33        for (j = 0; j < n; j++) {
34           hm[i*n+j] = t++;
35        }
36     }
37     // allocate space for device matrix
38     cudaMalloc((void **)&dm,msize);
39     // copy host matrix to device matrix
40     cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
41     // allocate host, device rowsum arrays
42     int rssize = n * sizeof(int);
43     hrs = (int *) malloc(rssize);
44     cudaMalloc((void **)&drs,rssize);
45     // set up parameters for threads structure
46     dim3 dimGrid(n,1); // n blocks
47     dim3 dimBlock(1,1,1); // 1 thread per block
48     // invoke the kernel
49     find1elt<<<dimGrid,dimBlock>>>(dm,drs,n);
50     // wait for kernel to finish
51     cudaThreadSynchronize();
52     // copy row vector from device to host
53     cudaMemcpy(hrs,drs,rssize,cudaMemcpyDeviceToHost);
54     // check results
55     if (n < 10) for(int i=0; i<n; i++) printf("%d\n",hrs[i]);
56     // clean up
57     free(hm);
58     cudaFree(dm);
59     free(hrs);
60     cudaFree(drs);
61  }
```

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**. They are called by the host and run on the device, thus serving 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.

```
1  __device__ int sumvector(float *x, int n)
```

  Note that unlike kernel functions, device functions can have return values, e.g. **int** above.
- When a kernel is called, each thread runs it. Each thread receives the same arguments.

- Each block and thread has an ID, stored in 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()** on the host to dynamically allocate space on the device's memory.[①] 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()**. The fourth argument specifies the direction, e.g. cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost or cudaMemcpyDeviceToDevice.

- Kernels return **void** values, so values are returned via a kernel's arguments.

- Device functions (which we don't have here) can return values. They are called only by kernel functions or other device functions.

- 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. In other words, this is barrier synchronization within blocks.

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

[①] This function cannot be called from the device itself. However, **malloc()** is available from the device, and device memory allocated by it can be copied to the host. See the NVIDIA programming guide for details.

```
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, in order to access the results of the call.

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). Since each SM is essentially a multicore machine in its own right, you might say the GPU is a multi-multiprocessor machine.

Each SM consists of a number of **streaming processors** (SPs), individual cores. The cores run threads, as with ordinary cores, but threads in an SM run in lockstep, to be explained below.

It is important to understand the motivation for this SM/SP 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 were originally in single precision only, though newer devices are capable of double 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.

SIMT Architecture

When you write a CUDA application program, you partition the threads 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.

The central 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).

The syntactic details of grid and block configuration will be presented in Section 5.3.4.

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." (On the other hand, threads in the same block but in different warps can diverge with no problem.)

"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.

The "hardware OS" runs largely in analogy with an ordinary OS:

- A process in an ordinary 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.
- With an ordinary 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, though, the analogous situation occurs when there is a long memory operation, to global memory; if a a warp of threads needs to access global memory (including local memory; see below), 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, quite a contrast to the OS case. Moreover, as noted above, the long latency of global memory may be solvable by having a lot of threads that the hardware can timeshare to hide that latency; while one warp is fetching data from memory, another warp can be executing, thus not losing time due to the long fetch delay. For these reasons, CUDA programmers typically employ a large number of threads, each of which does only a small amount of work—again, quite a contrast to something like OpenMP.

5.3.3 Memory Structure

The GPU memory hierarchy plays a key role in performance. Let's discuss the most important two types of memory first—shared and global.

Shared and Global Memory

Here is a summary:

type	shared	global
scope	glbl. to block	glbl. to app.
size	small	large
location	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 among themselves, 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 (details below).

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

  On the other hand, global memory is off-chip and very slow, taking hundreds of clock cycles per access instead of just a few. As noted earlier, this can be ameliorated by exploiting latency hiding; we will elaborate on this in Section 5.3.3.

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.* Data will start out in global memory, but if a variable is to be accessed multiple times by the GPU code, it's probably 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 eventually transmitted back to the host, the programmer must include code to copy it back to global memory.

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* to run contemporaneously by the hardware OS, first one half-warp and then the other.

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

```
1 __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**.

Shared memory consistency (recall Section 3.6) 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 in a block until **___syncthreads()** is called. On the other hand, writes by a thread *will* be visible to that same thread in subsequent reads without calling **___syncthreads()**. 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.

In the code fragment above, we allocated the shared memory through a C-style declaration:

```
__shared__ int abcsharedmem[100];
```

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

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

// CUDA example: illustrates kernel-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, with each thread copying one
   // element
   sv[me] = 2 * dv[me];
   dv[me] = sv[me];
}

int main(int argc, char **argv)
{
    int n = atoi(argv[1]); // number of matrix rows/cols
    int *hv, // host array
        *dv; // device array
    int vsize = n * sizeof(int); // size of array in bytes
```

```
25    // allocate space for host array
26    hv = (int *) malloc(vsize);
27    // fill test array with consecutive integers
28    int t = 0,i;
29    for (i = 0; i < n; i++)
30       hv[i] = t++;
31    // allocate space for device array
32    cudaMalloc((void **)&dv,vsize);
33    // copy host array to device array
34    cudaMemcpy(dv,hv,vsize,cudaMemcpyHostToDevice);
35    // set up parameters for threads structure
36    dim3 dimGrid(1,1);
37    dim3 dimBlock(n,1,1); // all n threads in the same block
38    // invoke the kernel; third argument is amount of shared memory
39    doubleit<<<dimGrid,dimBlock,vsize>>>(dv,n);
40    // wait for kernel to finish
41    cudaThreadSynchronize();
42    // copy row array from device to host
43    cudaMemcpy(hv,dv,vsize,cudaMemcpyDeviceToHost);
44    // check results
45    if (n < 10) for(int i=0; i<n; i++) printf("%d\n",hv[i]);
46    // clean up
47    free(hv);
48    cudaFree(dv);
49 }
```

Here the variable **sv** is kernel 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 region of space in this manner. This has two implications:

- Suppose we have two **___device___** functions, each declared an **extern ___shared___** array like this. Those two arrays will occupy the same place in memory!
- Suppose within one **___device___** function, we wish to have two **extern ___shared___** arrays. We cannot do that literally, but we can share the space via subarrays, e.g.:

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

  would set up **x** as a subarray of **sv** above, starting at element 120.

One can also set up shared arrays of fixed length in the same code. Declare them before the variable-length one.

In our example above, the array **sv** is syntactically local to the function **doubleit()**, but is shared by all invocations of that function in the block, thus acting "global" to them in a sense. But the point is that it is not accessible from within *other* functions running in that block. In order to achieve the latter situation, a shared array can be declared outside any function.

Global-Memory Performance Issues

As noted, the latency (Section **??**) 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 hardware will schedule another warp to run while the first is waiting for the memory access to complete. This is an example of a common parallel processing technique called **latency hiding**.

Second, the bandwidth (Section **??**) to global memory 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 works because the memory is low-order interleaved (Section 3.2.1), and is true for both reads and writes.

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

The programmer may be able to take advantage of coalescing, by a judicious choice of algorithms and/or by inserting padding into arrays (Section 3.2.2).

Shared-Memory Performance Issues

Shared memory is divided into banks, in a low-order interleaved manner (recall Section 3.2): 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. (Actually, older devices have 16 banks, while newer ones have 32.) 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, just as for the case of global memory.

An exception occurs in **broadcast**. If all threads in the block wish to read from the same word in the same bank, the word will be sent to all the requestors simultaneously without conflict. However, if only some theads try to read the same word, there may or may not be a conflict, as the hardware chooses a bank for broadcast in some unspecified way.

As in the discussion of global memory above, we should write our code to take advantage of these structures.

The biggest performance issue with shared memory is its size, as little as 16K per SM in many GPU cards. And remember, this is divvied up among the blocks on a given SM. If we have 4 blocks running on an SM, each one can only use 16K/4 = 4K bytes of shared memory.

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, meaning that it cannot be swapped out by the OS' virtual memory system. This allows the use of DMA hardware to do the memory copy, said to make **cudaMemcpy()** twice as fast.

Other Types of Memory

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

type	registers	local	constant	texture
scope	single thread	single thread	glbl. to app.	glbl. to app.
location	device	device	host+device cache	host+device cache
speed	fast	molasses	fast if cache hit	fast if cache hit
lifetime	kernel	kernel	application	application
host access?	no	no	yes	yes
device access?	read/write	read/write	read	read

- **Registers:**

  Each SM has a set of registers, 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

```
1 int z[20],i;
2 ...
3 y = z[i];
```

  Since registers are not indexable by the hardware, the compiler cannot allocate **z** to registers in this case. 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 be changed by device code. It is off-chip, thus potentially slow, 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()**, whose (simple form for the) call is

```
1 cudaMemcpyToSymbol(var_name,pointer_to_source,number_bytes_copy,cudaMemcpyHostToDevice)
```

For example:

```
__constant__ int x;  // not contained in any function

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

// device code
int z;
z = x;
```

Note again that the name Constant refers to the fact that device code cannot change it. But host code certainly can change it between kernel calls. This might be useful in iterative algorithms like this:

```
/ host code
for 1 to number of iterations
   set Constant array x
   call kernel (do scatter op)
   cudaThreadSynchronize()
   do gather op, using kernel results to form new x

// device code
use x together with thread-specific data
return results to host
```

- **Texture:**
  This is similar to constant memory, in the sense that it is read-only and cached. The difference is that the caching is two-dimensional. The elements **a[i][j]** and **a[i+1][j]** are far from each other in the global memory, but since they are "close" in a two-dimensional sense, they may reside in the same cache line.

5.3.4 Threads Hierarchy

Following 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 first 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 ($1 \times 1 \times 1$) thread.

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 of type **dim3**, in this case our variables **gridDim** and **blockDim**, accessible to the programmer, again with member variables for the various dimensions, e.g. **blockDim.x** for the size of the X dimension for the number of threads per block.

- 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. If for instance one is programming computation of heat flow across a two-dimensional slab, the programmer may find it clearer to use two-dimensional IDs for the threads. *But this does not correspond to any physical arrangement in the hardware.*

As noted, 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 12.2. In a small example there, the matrix

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{5.1}$$

is partitioned as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \tag{5.2}$$

where

$$A_{00} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{5.3}$$

$$A_{01} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{5.4}$$

$$A_{10} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{5.5}$$

and

$$A_{11} = \begin{pmatrix} 2 \end{pmatrix}. \tag{5.6}$$

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, Within and Between Blocks

As mentioned 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 (from both threads) in order to get the latest value. Keep in mind that within a block, different warps will run at different times, making synchronization vital.

Remember too that threads across blocks cannot sync with each other in this manner. There are, though, several **atomic** operations—read/modify/write actions that a thread can execute without **pre-emption**, i.e. without interruption—available on both global and shared memory. For example, **atomicAdd()** performs a fetch-and-add operation, as described in Section 3.4.3 of this book. The call is

```
1 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. The return value of the function is the value originally at that address before the operation.

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, e.g.

```
1 nvcc -g -G yoursrc.cu -arch=sm_11
```

Though a barrier could in principle be constructed from the atomic operations, its overhead would be quite high. In earlier models that was near a microsecond, and though that problem has been ameliorated in more recent models, implementing a barrier in this manner . would not be not much faster than attaining interblock synchronization by returning to the host and calling **cudaThreadSynchronize()** there. Recall that the latter *is* a possible way to implement a barrier, since global memory stays intact in between kernel calls, but again, it would be slow.

So, what if synchronization is really needed? This is the case, for instance, for iterative algorithms, where all threads must wait at the end of each iteration.

If you have a small problem, maybe you can get satisfactory performance by using just one block. You'll have to use a larger granularity, i.e. more work assigned to each thread. But using just one block means you're using only one SM, thus only a fraction of the potential power of the machine.

If you use multiple blocks, though, your only feasible option for synchronization is to rely on returns to the host, where synchronization occurs via **cudaThreadSynchronize()**. You would then have the situation outlined in the discussion of Constant memory in Section 5.3.3.

5.5 More on the Blocks/Threads Tradeoff

Resource size considerations must be kept in mind when you design your code and your grid configuration. In particular, note the following:

- Each block in your code is assigned to some SM. It will be tied to that SM during the entire execution of your kernel, though of course it will not constantly be running during that time.
- If there are more blocks than can be accommodated by all the SMs, then some blocks will need to wait for assignment; when a block finishes, that block's resources, e.g. shared memory, can now be assigned to a waiting block.
- The programmer has no control over which block is assigned to which SM.
- Within a block, threads execute by the warp, 32 threads. At any give time, the SM is running one warp, chosen by the GPU OS.
- The GPU has a limit on the number of threads that can run on a single block, typically 512, and on the total number of threads running on an SM, 786.
- If a block contains fewer than 32 threads, only part of the processing power of the SM it's running on will be used. So block size should normally be at least 32. Moreover, for the same reason, block size should ideally be a multiple of 32.
- If your code makes used of shared memory, or does within-block synchronization, the larger the block size, the better.
- We want to use the full power of the GPU, with its many SMs, thus implying a need to use at least as many blocks as there are SMs (which may require smaller blocks).
- Moreover, due to the need for latency hiding in memory access, we want to have lots of warps, so that some will run while others are doing memory access.
- Two threads doing unrelated work, or the same work but with many if/elses, would cause a lot of thread divergence if they were in the same block.
- A commonly-cited rule of thumb is to have between 128 and 256 threads per block.

Though there is a limit on the number of blocks, this limit will be much larger than the number of SMs. So, you may have multiple blocks running on the same SM. Since execution is scheduled by the warp anyway, there appears to be no particular drawback to having more than one block on the same SM.

5.6 Hardware Requirements, Installation, Compilation, Debugging

You do need a suitable NVIDIA video card. There is a list at `http://www.nvidia.com/object/cuda_gpus.html`; see also the Wikipedia entry, `http://en.wikipedia.org/wiki/CUDA#Supported_GPUs`. If you have a Linux system, run **lspci** to determine what kind you have.[②]

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
```

The **-g -G** options are for setting up debugging, the first for host code, the second for device code. You may also need to specify

```
-I/your_CUDA_include_path
```

to pick up the file **cuda.h**. Run the code as you normally would.

You may need to take special action 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_g5aa4f47938af8276f08074d0`
`html`.

Here's a simple program to check some of the properties of device 0:

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

int main()
{
   cudaDeviceProp Props;
   cudaGetDeviceProperties( &Props,0);

   printf("shared mem: %d)\n", Props.sharedMemPerBlock);
   printf("max threads/block: %d\n",Props.maxThreadsPerBlock);
   printf("max blocks: %d\n",Props.maxGridSize[0]);
   printf("total Const mem: %d\n",Props.totalConstMem);
}
```

[②] It might be in **/sbin**.

Under older versions of CUDA, such as 2.3, one can debug using GDB as usual. You must compile your program in emulation mode, using the **-deviceemu** command-line option. This is no longer available as of version 3.2. CUDA also includes a special version of GDB, CUDA-GDB (invoked as **cuda-gdb**) for real-time debugging. However, on Unix-family platforms it runs only if X11 is not running. Short of dedicating a machine for debugging, you may find it useful to install a version 2.3 in addition to the most recent one to use for debugging.

5.7 Example: Improving the Row Sums Program

The issues involving coalescing in Section 5.3.3 would suggest that our rowsum code might run faster with column sums, to take advantage of the memory banking. (So the user would either need to take the transpose first, or have his code set up so that the matrix is in transpose form to begin with.) 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.[③]

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.

[③] Neither has the CPU-only version of the program been optimized. As pointed out by Bill Hsu, the row-major version of that program should run faster than the column-major one, due to cache consideration.

5.8 Example: Finding the Mean Number of Mutual Outlinks

As in Sections 2.4.3 and 4.12, 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.

```
1  #include <cuda.h>
2  #include <stdio.h>
3
4  // CUDA example: finds mean number of mutual outlinks, among all pairs
5  // of Web sites in our set; in checking all (i,j) pairs, thread k will
6  // handle all i such that i mod totth = k, where totth is the number of
7  // threads
8
9  // procpairs() processes all pairs for a given thread
10 __global__ void procpairs(int *m, int *tot, int n)
11 { int totth = gridDim.x * blockDim.x, // total number of threads
12       me = blockIdx.x * blockDim.x + threadIdx.x; // my thread number
13   int i,j,k,sum = 0;
14   for (i = me; i < n; i += totth) { // do various rows i
15      for (j = i+1; j < n; j++) { // do all rows j > i
16         for (k = 0; k < n; k++)
17            sum += m[n*i+k] * m[n*j+k];
18      }
19   }
20   atomicAdd(tot,sum);
21 }
22
23 int main(int argc, char **argv)
24 { int n = atoi(argv[1]), // number of vertices
25       nblk = atoi(argv[2]); // number of blocks
26    int *hm, // host matrix
27        *dm, // device matrix
28        htot, // host grand total
29        *dtot; // device grand total
30    int msize = n * n * sizeof(int); // size of matrix in bytes
31    // allocate space for host matrix
32    hm = (int *) malloc(msize);
33    // as a test, fill matrix with random 1s and 0s
34    int i,j;
35    for (i = 0; i < n; i++) {
```

```
36       hm[n*i+i] = 0;
37       for (j = 0; j < n; j++) {
38          if (j != i) hm[i*n+j] = rand() % 2;
39       }
40    }
41    // allocate space for device matrix
42    cudaMalloc((void **)&dm,msize);
43    // copy host matrix to device matrix
44    cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
45    htot = 0;
46    // set up device total and initialize it
47    cudaMalloc((void **)&dtot,sizeof(int));
48    cudaMemcpy(dtot,&htot,sizeof(int),cudaMemcpyHostToDevice);
49    // set up parameters for threads structure
50    dim3 dimGrid(nblk,1);
51    dim3 dimBlock(192,1,1);
52    // invoke the kernel
53    procpairs<<<dimGrid,dimBlock>>>(dm,dtot,n);
54    // wait for kernel to finish
55    cudaThreadSynchronize();
56    // copy total from device to host
57    cudaMemcpy(&htot,dtot,sizeof(int),cudaMemcpyDeviceToHost);
58    // check results
59    if (n <= 15) {
60       for (i = 0; i < n; i++) {
61          for (j = 0; j < n; j++)
62             printf("%d ",hm[n*i+j]);
63          printf("\n");
64       }
65    }
66    printf("mean = %f\n",htot/float((n*(n-1))/2));
67    // clean up
68    free(hm);
69    cudaFree(dm);
70    cudaFree(dtot);
71 }
```

Again we've used the method in Section 2.4.3 to partition the various pairs (i,j) to the different threads. Note the use of **atomicAdd()**.

The above code is hardly optimal. The reader is encouraged to find improvements.

5.9 Example: Finding Prime Numbers

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <cuda.h>
4
5  // CUDA example: illustration of shared memory allocation at run time;
6  // finds primes using classical Sieve of Erathosthenes: make list of
7  // numbers 2 to n, then cross out all multiples of 2 (but not 2 itself),
8  // then all multiples of 3, etc.; whatever is left over is prime; in our
9  // array, 1 will mean "not crossed out" and 0 will mean "crossed out"
10
11 // IMPORTANT NOTE: uses shared memory, in a single block, without
12 // rotating parts of array in and out of shared memory; thus limited to
13 // n <= 4000 if have 16K shared memory
14
15 // initialize sprimes, 1s for the odds, 0s for the evens; see sieve()
16 // for the nature of the arguments
17 __device__ void initsp(int *sprimes, int n, int nth, int me)
18 {
19    int chunk,startsetsp,endsetsp,val,i;
20    sprimes[2] = 1;
21    // determine sprimes chunk for this thread to init
22    chunk = (n-1) / nth;
23    startsetsp = 2 + me*chunk;
24    if (me < nth-1) endsetsp = startsetsp + chunk - 1;
25    else endsetsp = n;
26    // now do the init
27    val = startsetsp % 2;
28    for (i = startsetsp; i <= endsetsp; i++) {
29       sprimes[i] = val;
30       val = 1 - val;
31    }
32    // make sure sprimes up to date for all
33    __syncthreads();
34 }
35
36 // copy sprimes back to device global memory; see sieve() for the nature
37 // of the arguments
38 __device__ void cpytoglb(int *dprimes, int *sprimes, int n, int nth, int me)
39 {
```

```
40    int startcpy,endcpy,chunk,i;
41    chunk = (n-1) / nth;
42    startcpy = 2 + me*chunk;
43    if (me < nth-1) endcpy = startcpy + chunk - 1;
44    else endcpy = n;
45    for (i = startcpy; i <= endcpy; i++) dprimes[i] = sprimes[i];
46    __syncthreads();
47 }
48
49 // finds primes from 2 to n, storing the information in dprimes, with
50 // dprimes[i] being 1 if i is prime, 0 if composite; nth is the number
51 // of threads (threadDim somehow not recognized)
52 __global__ void sieve(int *dprimes, int n, int nth)
53 {
54    extern __shared__ int sprimes[];
55    int me = threadIdx.x;
56    int nth1 = nth - 1;
57    // initialize sprimes array, 1s for odds, 0 for evens
58    initsp(sprimes,n,nth,me);
59    // "cross out" multiples of various numbers m, with each thread doing
60    // a chunk of m's; always check first to determine whether m has
61    // already been found to be composite; finish when m*m > n
62    int maxmult,m,startmult,endmult,chunk,i;
63    for (m = 3; m*m <= n; m++) {
64       if (sprimes[m] != 0) {
65          // find largest multiple of m that is <= n
66          maxmult = n / m;
67          // now partition 2,3,...,maxmult among the threads
68          chunk = (maxmult - 1) / nth;
69          startmult = 2 + me*chunk;
70          if (me < nth1) endmult = startmult + chunk - 1;
71          else endmult = maxmult;
72       }
73       // OK, cross out my chunk
74       for (i = startmult; i <= endmult; i++) sprimes[i*m] = 0;
75    }
76    __syncthreads();
77    // copy back to device global memory for return to host
78    cpytoglb(dprimes,sprimes,n,nth,me);
79 }
80
81 int main(int argc, char **argv)
82 {
```

```
83    int n = atoi(argv[1]), // will find primes among 1,...,n
84        nth = atoi(argv[2]); // number of threads
85    int *hprimes, // host primes list
86        *dprimes; // device primes list
87    int psize = (n+1) * sizeof(int); // size of primes lists in bytes
88    // allocate space for host list
89    hprimes = (int *) malloc(psize);
90    // allocate space for device list
91    cudaMalloc((void **)&dprimes,psize);
92    dim3 dimGrid(1,1);
93    dim3 dimBlock(nth,1,1);
94    // invoke the kernel, including a request to allocate shared memory
95    sieve<<<dimGrid,dimBlock,psize>>>(dprimes,n,nth);
96    // check whether we asked for too much shared memory
97    cudaError_t err = cudaGetLastError();
98    if(err != cudaSuccess) printf("%s\n",cudaGetErrorString(err));
99    // wait for kernel to finish
100   cudaThreadSynchronize();
101   // copy list from device to host
102   cudaMemcpy(hprimes,dprimes,psize,cudaMemcpyDeviceToHost);
103   // check results
104   if (n <= 1000) for(int i=2; i<=n; i++)
105      if (hprimes[i] == 1) printf("%d\n",i);
106   // clean up
107   free(hprimes);
108   cudaFree(dprimes);
109 }
```

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 1.3.2, 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. Moreover, by using just one block, we are only using a small part of the CPU. Extending the program to work for larger values of **n** would require some careful planning if we still wish to use shared memory.

5.10 Example: Finding Cumulative Sums

Here we wish to compute cumulative sums. For instance, if the original array is (3,1,2,0,3,0,1,2), then it is changed to (3,4,6,6,9,9,10,12).

(Note: This is a special case of the *prefix scan* problem, covered in Chapter **??**.)

The general plan is for each thread to operate on one chunk of the array. A thread will find cumulative sums for its chunk, and then adjust them based on the high values of the chunks that precede it. In the above example, for instance, say we have 4 threads. The threads will first produce (3,4), (2,2), (3,3) and (1,3). Since thread 0 found a cumulative sum of 4 in the end, we must add 4 to each element of (2,2), yielding (6,6). Thread 1 had found a cumulative sum of 2 in the end, which together with the 4 found by thread 0 makes 6. Thus thread 2 must add 6 to each of its elements, i.e. add 6 to (3,3), yielding (9,9). The case of thread 3 is similar.

Below is code for the special case of a single block:

```
1  // for this simple illustration, it is assumed that the code runs in
2  // just one block, and that the number of threads evenly divides n
3
4  // improvements that could be made:
5  // 1. change to multiple blocks, to try to use all SMs
6  // 2. possibly use shared memory
7  // 3. have each thread work on staggered elements of dx, rather than
8  // on contiguous ones, to get more efficient bank access
9
10 #include <cuda.h>
11 #include <stdio.h>
12
13 __global__ void cumulker(int *dx, int n)
14 {
15    int me = threadIdx.x;
16    int csize = n / blockDim.x;
17    int start = me * csize;
18    int i,j,base;
19    for (i = 1; i < csize; i++) {
20       j = start + i;
```

```
21       dx[j] = dx[j-1] + dx[j];
22    }
23    __syncthreads();
24    if (me > 0) {
25       base = 0;
26       for (j = 0; j < me; j++)
27          base += dx[(j+1)*csize-1];
28    }
29    __syncthreads();
30    if (me > 0) {
31       for (i = start; i < start + csize; i++)
32          dx[i] += base;
33    }
34 }
```

5.11 When Is it Advantageous to Use Shared Memory

Shared memory only helps if we are doing multiple accesses to the data. If for instance our code does a single read and a single write to an element of an array, then transferring it back and forth between global and shared memory isn't worthwhile.

Would the cumulative-sums program in Section 5.10 benefit from the use of shared memory? (Put aside the fact that the code runs in just one block, making use of just a sliver of the machine.) The answer appears to be that a modest imrprovement might be obtained. Each thread (except the first) reads many elements of **dx** twice, some of them three times. There are also writes.

The case of the prime-finder program in Section 5.9 is less clear, and probably quite dependent on whether we are using the more advanced GPUs, which feature at least some L1 cache space.

5.12 Example: Transforming an Adjacency Matrix

Here is a CUDA version of the code in Section 4.13.

```
1  // CUDA example
2
3  // takes a graph adjacency matrix for a directed graph, and converts it
4  // to a 2-column matrix of pairs (i,j), meaning an edge from vertex i to
5  // vertex j; the output matrix must be in lexicographical order
6
7  // not claimed efficient, either in speed or in memory usage
8
9  #include <cuda.h>
10 #include <stdio.h>
11
12 // needs -lrt link flag for C++
```

```
13 #include <time.h>
14 float timediff(struct timespec t1, struct timespec t2)
15 { if (t1.tv_nsec > t2.tv_nsec) {
16         t2.tv_sec -= 1;
17         t2.tv_nsec += 1000000000;
18    }
19    return t2.tv_sec-t1.tv_sec + 0.000000001 * (t2.tv_nsec-t1.tv_nsec);
20 }
21
22
23 // kernel transgraph() does this work
24 // arguments:
25 // adjm: the adjacency matrix (NOT assumed symmetric), 1 for edge, 0
26 // otherwise; note: matrix is overwritten by the function
27 // n: number of rows and columns of adjm
28 // adjmout: output matrix
29 // nout: number of rows in adjmout
30
31 __global__ void tgkernel1(int *dadjm, int n, int *dcounts)
32 { int tot1s,j;
33    int me = blockDim.x * blockIdx.x + threadIdx.x;
34    tot1s = 0;
35    for (j = 0; j < n; j++) {
36       if (dadjm[n*me+j] == 1) {
37          dadjm[n*me+tot1s++] = j;
38       }
39    dcounts[me] = tot1s;
40    }
41 }
42
43 __global__ void tgkernel2(int *dadjm, int n,
44    int *dcounts, int *dstarts, int *doutm)
45 { int outrow,num1si,j;
46    // int me = threadIdx.x;
47    int me = blockDim.x * blockIdx.x + threadIdx.x;
48    // fill in this thread's portion of doutm
49    outrow = dstarts[me];
50    num1si = dcounts[me];
51    if (num1si > 0) {
52       for (j = 0; j < num1si; j++) {
53          doutm[2*outrow+2*j] = me;
54          doutm[2*outrow+2*j+1] = dadjm[n*me+j];
55       }
```

```
56    }
57 }
58
59 // replaces counts by cumulative counts
60 void cumulcounts(int *c, int *s, int n)
61 { int i;
62    s[0] = 0;
63    for (i = 1; i < n; i++) {
64       s[i] = s[i-1] + c[i-1];
65    }
66 }
67
68 int *transgraph(int *hadjm, int n, int *nout, int gsize, int bsize)
69 { int *dadjm; // device adjacency matrix
70    int *houtm; // host output matrix
71    int *doutm; // device output matrix
72    int *hcounts; // host counts vector
73    int *dcounts; // device counts vector
74    int *hstarts; // host starts vector
75    int *dstarts; // device starts vector
76    hcounts = (int *) malloc(n*sizeof(int));
77    hstarts = (int *) malloc(n*sizeof(int));
78    cudaMalloc((void **)&dadjm,n*n*sizeof(int));
79    cudaMalloc((void **)&dcounts,n*sizeof(int));
80    cudaMalloc((void **)&dstarts,n*sizeof(int));
81    houtm = (int *) malloc(n*n*sizeof(int));
82    cudaMalloc((void **)&doutm,n*n*sizeof(int));
83    cudaMemcpy(dadjm,hadjm,n*n*sizeof(int),cudaMemcpyHostToDevice);
84    dim3 dimGrid(gsize,1);
85    dim3 dimBlock(bsize,1,1);
86    // calculate counts and starts first
87    tgkernel1<<<dimGrid,dimBlock>>>(dadjm,n,dcounts);
88    // cudaMemcpy(hadjm,dadjm,n*n*sizeof(int),cudaMemcpyDeviceToHost);
89    cudaMemcpy(hcounts,dcounts,n*sizeof(int),cudaMemcpyDeviceToHost);
90    cumulcounts(hcounts,hstarts,n);
91    *nout = hstarts[n-1] + hcounts[n-1];
92    cudaMemcpy(dstarts,hstarts,n*sizeof(int),cudaMemcpyHostToDevice);
93    tgkernel2<<<dimGrid,dimBlock>>>(dadjm,n,dcounts,dstarts,doutm);
94    cudaMemcpy(houtm,doutm,2*(*nout)*sizeof(int),cudaMemcpyDeviceToHost);
95    free(hcounts);
96    free(hstarts);
97    cudaFree(dadjm);
98    cudaFree(dcounts);
```

```
99    cudaFree(dstarts);
100   return houtm;
101 }
102
103 int main(int argc, char **argv)
104 { int i,j;
105   int *adjm; // host adjacency matrix
106   int *outm; // host output matrix
107   int n = atoi(argv[1]);
108   int gsize = atoi(argv[2]);
109   int bsize = atoi(argv[3]);
110   int nout;
111   adjm = (int *) malloc(n*n*sizeof(int));
112   for (i = 0; i < n; i++)
113      for (j = 0; j < n; j++)
114         if (i == j) adjm[n*i+j] = 0;
115         else adjm[n*i+j] = rand() % 2;
116   if (n < 10) {
117      printf("adjacency matrix: \n");
118      for (i = 0; i < n; i++) {
119         for (j = 0; j < n; j++) printf("%d ",adjm[n*i+j]);
120         printf("\n");
121      }
122   }
123
124   struct timespec bgn,nd;
125   clock_gettime(CLOCK_REALTIME, &bgn);
126
127   outm = transgraph(adjm,n,&nout,gsize,bsize);
128   printf("num rows in out matrix = %d\n",nout);
129   if (nout < 50) {
130      printf("out matrix: \n");
131      for (i = 0; i < nout; i++)
132         printf("%d %d\n",outm[2*i],outm[2*i+1]);
133   }
134
135   clock_gettime(CLOCK_REALTIME, &nd);
136   printf("%f\n",timediff(bgn,nd));
137 }
```

5.13 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.

5.14 Loop Unrolling

Loop unrolling is an old technique used on uniprocessor machines to achieve speedup due to branch elimination and the like. Branches make it difficult to do instruction or data prefetching, so eliminating them may speed things up.

The CUDA compiler provides the programmer with the **unroll** pragma to request loop unrolling. Here an n-iteration **for** loop is changed to k copies of the body of the loop, each working on about n/k iterations. If n and k are known constant, GPU registers can be used to implement the unrolled loop.

For example, the loop

```
for (i = 0; i < 2; i++ {
   sum += x[i];
   sum2 += x[i]*x[i];
}
```

could be unrolled to

```
sum += x[1];
sum2 += x[1]*x[1];
sum += x[2];
sum2 += x[2]*x[2];
```

Here n = k = 2. If **x** is local to this function, then unrolling will allow the compiler to store it in a register, which could be a great performance enhancer.

The compiler will try to do loop unrolling even if the programmer doesn't request it, but the programmer can try to control things by using the pragma:

```
#pragma unroll k
```

suggest to the compiler a k-fold unrolling. Setting k = 1 will instruct the compiler *not* to unroll.

5.15 Short Vectors

In CUDA, there are types such as **int4**, **char2** and so on, up to four elements each. So, an **uint4** type is a set of four unsigned **int**s. These are called **short vectors**.

The key point is that a short vector can be treated as a single word in terms of memory access and GPU instructions. It may be possible to reduce time by a factor of 4 by dividing arrays into chunks of four contiguous words and making short vectors from them.

5.16 The New Generation

The latest GPU architecture from NVIDIA is called Kepler. Many of the advances are of the "bigger and faster than before" type. These are important, but be sure to note the significant architectural changes, including:

- Host memory, device global memory and device shared memory share a unifed address space.
- On-chip memory can be apportioned to both shared memory and cache memory. Since shared memory is in essence a programmer-managed cache, this gives the programmer access to a real cache, a great convenience to the programmer though with a possible sacrifice in speed. Note by the way that this cache is aimed at spatial locality, not temporal locality.

5.17 CUDA from a Higher Level

CUDA programming can involve a lot of work, and one is never sure that one's code is fully efficient. Fortunately, a number of libraries of tight code have been developed for operations that arise often in parallel programming.

You are of course using CUDA code at the bottom, but without explicit kernel calls. And again, remember, the contents of device global memory are persistent across kernel calls in the same application. Therefore you can mix explicit CUDA code and calls to these libraries. Your program might have multiple kernel invocations, some CUDA and others to the libraries, with each using data in device global memory that was written by earlier kernels. In some cases, you may need to do a conversion to get the proper type.

These packages can be deceptively simple. **Remember, each call to a function in these packages involves a CUDA kernel call—with the associated overhead.**

Programming in these libraries is typically much more convenient than in direct CUDA. Note, though, that even though these libraries have been highly optimized for what they are intended to do, they will not generally give you the fastest possible code for any given CUDA application.

We'll discuss a few such libraries in this section.

5.17.1 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 directly programming in CUDA.

Example: Row Sums Once Again

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 if you are mixing CUDA and CUBLAS, you would use **nvcc**:

```
nvcc -g -G 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 ("mv")
   cublasSgemv('n',n,n,1.0,dm,n,drs,1,0.0,drs,1);
   // copy result back to host
```

```
35    cublasGetVector(n,sizeof(float),drs,1,hrs,1);
36    // check results
37    if (n < 20) for (i = 0; i < n; i++) printf("%f\n",hrs[i]);
38    // clean up on device (should call free() on host too)
39    cublasFree(dm);
40    cublasFree(drs);
41    cublasShutdown();
42 }
```

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

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

```
1 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.

The call

```
1 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** dimensioned rows. This seems redundant, but this is needed in cases of matrix tiling, where the number of rows of a tile would be less than the number of rows of the matrix as a whole.

The 1s in the call

```
1 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 any particular row of interest would be spaced 500 elements apart, again keeping in mind that column-major order is assumed.

The actual matrix multiplication is done here:

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

The "mv" in "cublasSgemv" stands for "matrix times vector." Here the call says: no ('n'), we do not want the matrix to be transposed; the matrix has **n** rows and **n** columns; we wish the matrix to be multiplied by 1.0 (if 0, the multiplication is not actually performed, which we could have here); the matrix is at **dm**; the number of dimensioned rows of the matrix is **n**; the vector is at **drs**; the elements of the vector are spaced 1 word apart; we wish the vector to not be multiplied by a scalar (see note above); the resulting vector will be stored at **drs**, 1 word apart.

Further information is available in the CUBLAS manual.

5.17.2 Thrust

The Thrust library is usable not only with CUDA but also to general OpenMP code! So I've put my coverage of Thrust in a separate chapter, Chapter 6.

5.17.3 CUDPP

CUDPP is similar to Thrust (though CUDPP was developed earlier) in terms of operations offered. It is perhaps less flexible than Thrust, but is easier to learn and is said to be faster.

(No examples yet, as the author did not have access to a CUDPP system yet.)

5.17.4 CUFFT

CUFFT does for the Fast Fourier Transform what CUBLAS does for linear algebra, i.e. it provides CUDA-optimized FFT routines.

5.18 Other CUDA Examples in This Book

There are additional CUDA examples in later sections of this book. These include:[4]

- Prof. Richard Edgar's matrix-multiply code, optimized for use of shared memory, Section 12.3.2.
- Odd/even transposition sort, Section 13.3.3, showing a typical CUDA pattern for iterative algorithms.
- Gaussian elimination for linear systems, Section 12.5.1.

[4] If you are reading this presentation on CUDA separately from the book, the book is at `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`

第 6 章　Introduction to Thrust Programming

In the spirit of CUBLAS (Section 5.17.1) and other packages, the CUDA people have brought in another package to ease CUDA programming, Thrust. It uses the C++ STL library as a model, and Thrust is indeed a C++ template library. It includes various data manipulation routines, such as for sorting and prefix scan operations.

6.1 Compiling Thrust Code

Thrust allows the programmer a choice of *back ends*, i.e. platforms on which the executable code will run. In addition to the CUDA back end, for running on the GPU, one can also choose OpenMP as the back end. The latter choice allows the high-level expressive power of Thrust to be used on multicore machines. A third choice is Intel's TBB language.

6.1.1 Compiling to CUDA

If your CUDA version is at least 4.0, then Thrust is included, which will be assumed here. In that case, you compile Thrust code with **nvcc**, no special link commands needed.

6.1.2 Compiling to OpenMP

You can use Thrust to generate OpenMP code. The Thrust "include" files work without having a GPU. Here for instance is how you would compile the first example program below:①

```
g++ -g -O2 -o unqcount unqcount.cpp -fopenmp -lgomp \
  -DTHRUST_DEVICE_BACKEND=THRUST_DEVICE_BACKEND_OMP \
  -I/usr/home/matloff/Tmp/tmp1
```

I had no CUDA-capable GPU on this machine, but put the Thrust include directory tree in **/usr/home/matloff/Tmp/tmp1**, and then compiled as with any other include file.

The result is real OpenMP code.② Everywhere you set up a Thrust vector, you'll be using OpenMP, i.e. the threads set up by Thrust will be OpenMP threads on the CPU rather than CUDA threads on the GPU.③ You set the number of threads as you do with any OpenMP program, e.g. with the environment variable OMP_NUM_THREADS.

①Note that we used a **.cpp** suffix for the source file name, instead of **.cu**. Or, we can use the **-x cu** if compiling with **nvcc**.
②If you search through the Thrust source code, you'll fine **omp** pragmas.
③Threads will not be set up if you use host arraysvectors.

6.2 Example: Counting the Number of Unique Values in an Array

As our first example, suppose we wish to determine the number of distinct values in an integer array. The following code may not be too efficient, but as an introduction to Thrust fundamental building blocks, we'll take the following approach:

(a) sort the array

(b) compare the array to a shifted version of itself, so that changes from one distinct element to another can be detected, producing an array of 1s (change) and 0s (no change)

(c) count the number of 1s

Here's the code:

```
1  // various Thrust includes
2  #include <thrust/host_vector.h>
3  #include <thrust/device_vector.h>
4  #include <thrust/generate.h>
5  #include <thrust/sort.h>
6  #include <thrust/copy.h>
7  #include <thrust/count.h>
8  #include <cstdlib>
9
10 int rand16() // generate random integers mod 16
11 { return rand() % 16; }
12
13 // C++ functor, to be called from thrust::transform(); compares
14 // corresponding elements of the arrays x and y, yielding 0 when they
15 // match, 1 when they don't
16 struct finddiff
17 {
18   __device__ int operator()(const int& x, const int&y)
19   { return x == y?0:1; }
20 };
21
22 int main(void)
23 {
24    // generate test data, 1000 random numbers, on the host, int type
25    thrust::host_vector<int> hv(1000);
26    thrust::generate(hv.begin(), hv.end(), rand16);
27
28    // copy data to the device, creating a vector there
29    thrust::device_vector<int> dv = hv;
30
```

```
31    // sort data on the device
32    thrust::sort(dv.begin(), dv.end());
33
34    // create device vector to hold differences, with length 1 less than
35    // dv's length
36    thrust::device_vector<int> diffs(dv.size()-1);
37
38    // find the diffs; note that the syntax is finddiff(), not finddiff
39    thrust::transform(dv.begin(),dv.end()-1,
40       dv.begin()+1,diffs.begin(),finddiff());
41
42    // count the 1s, by removing 0s and checking new length
43    // (or could use thrust::count())
44    int ndiffs = thrust::reduce(diffs.begin(), diffs.end(), (int) 0,
45       thrust::plus<int>());
46    printf("# distinct: %d\n",ndiffs+1);
47
48    // we've achieved our goal, but let's do a little more
49    // transfer data back to host
50    thrust::copy(dv.begin(), dv.end(), hv.begin());
51
52    printf("the sorted array:\n");
53    for (int i = 0; i < 1000; i++) printf("%d\n",hv[i]);
54
55    return 0;
56 }
```

After generating some random data on a host array **hv**, we copy it to the device, creating a vector **dv** there. This code is certainly much simpler to write than slogging through calls to **cudaMalloc()** and **cudaMemcpy()**!

The heart of the code is the call to **thrust::transform()**, which is used to implement step (b) in our outline above. It performs a "map" operation as in functional programming, taking one or two arrays (the latter is the case here) as input, and outputting an array of the same size.

This example, as is typical in Thrust code, defines a **functor**. Well, what is a functor? This is a C++ mechanism to produce a callable function, laregly similar in goal to using a pointer to a function. In the context above, we are turning a C++ struct into a callable function, and we can do so with classes too. Since structs and classes can have member variables, we can store needed data in them, and that is what distinguishes functors from function pointers.

The transform function does elementwise operation—it calls the functor on each corresponding pair of elements from the two input arguments (0th element with 0th element, 1st with 1st, etc.), placing the results in the output array. So we must thus design our functor to do the map operation. In this case, we want to compare successive elements of our array (after sorting it), so we must find

a way to do this through some element-by-element operation. The solution is to do elementwise comparison of the array and its shifted version. The call is

```
thrust::transform(dv.begin(),dv.end()-1,
      dv.begin()+1,diffs.begin(),finddiff());
```

Note the parentheses in "finddiff()." This is basically a constructor, creating an instance of a **finddiff** object and returning a pointer to it. By contrast, in the code

```
thrust::generate(hv.begin(), hv.end(), rand16);
```

rand16() is an ordinary function, not a functor, so we just write its name here, thus passing a pointer to the function.

In the code

```
__device__ int operator()(const int& x, const int&y)
{ return x == y?0:1; }
```

the C++ keyword **operator** says we are defining a function, which in this case has two **int** inputs and an **int** output. We stated earlier that functors are callable structs, and this is what gets called.

Thrust vectors have built-in member functions **begin()** and **end()**, that specify the start and the place 1 element past the end of the array. Note that we didn't actually create our shifted array in

```
thrust::transform(dv.begin(),dv.end()-1,
      dv.begin()+1,diffs.begin(),finddiff());
```

Instead, we specified "the array beginning 1 element past the start of **dv**."

The "places" returned by calling **begin()** and **end()** above are formally called **iterators**, and work in a manner similar to pointers. Note again that **end()** returns a pointer to the location just *after* the last element of the array. The pointers are of type **thrust::device_vector<int>::iterator** here, with similar expressions for cases other than **int** type.

The transform function, in this case the comparison operation, will be done in parallel, on the GPU or other backend, as was the sorting. All that's left is to count the 1s. We want to do that in parallel too, and Thrust provides another functional programming operation, reduction (as in OpenMP). We specify Thrust's built-in addition function (we could have defined our own if it were a more complex situation) as the operation, and 0 as the initial value:

```
int ndiffs = thrust::reduce(diffs.begin(), diffs.end(), (int) 0, thrust::plus<
   int>());
```

We also could have used Thrust's **thrust::count()** function for further convenience.

Below is a shorter version of our unique-values-counter program, using **thrust::unique()**. Note that that function only removes *consecutive* duplicates, so the preliminary sort is still needed.

```
// various Thrust includes
#include <thrust/host_vector.h>
```

```
3  #include <thrust/device_vector.h>
4  #include <thrust/generate.h>
5  #include <thrust/sort.h>
6  #include <thrust/copy.h>
7  #include <thrust/unique.h>
8  #include <cstdlib>
9
10 int rand16()
11 { return rand() % 16; }
12
13 int main(void)
14 {
15    thrust::host_vector<int> hv(1000);
16    thrust::generate(hv.begin(), hv.end(), rand16);
17    thrust::device_vector<int> dv = hv;
18    thrust::sort(dv.begin(), dv.end());
19    thrust::device_vector<int>::iterator newend =
20       thrust::unique(dv.begin(),dv.end());
21    printf("# distinct: %d\n",newend - dv.begin());
22    return 0;
23 }
```

Note the line

```
1  thrust::device_vector<int>::iterator newend =
2    thrust::unique(dv.begin(),dv.end());
```

The **unique()** function returns an iterator pointing to (one element past) the end of the result of applying the unique-ifying operation. We can then "subtract" iterator values to get our desired count:

```
1  printf("# distinct: %d\n",newend - dv.begin());
```

6.3 Example: A Plain-C Wrapper for Thrust sort()

We may wish to wrap utility Thrust code in a function callable from a purely C/C++ program. The code below does that for the Thrust sort function.

```
1  // definitely needed
2  extern "C" void tsort(int *x, int *nx);
3
4  #include <thrust/device_vector.h>
5  #include <thrust/sort.h>
6
7  // nx set up as pointer so can call from R
```

```
void tsort(int *x, int *nx)
{ int n = *nx;
  // set up device vector and copy x to it
  thrust::device_vector<int> dx(x,x+n);
  // sort, then copy back to x
  thrust::sort(dx.begin(), dx.end());
  thrust::copy(dx.begin(), dx.end(),x);
}
```

To compile in the CUDA case, run **nvcc -c** and then run **gcc** (or whatever) as usual, making sure to link with the CUDA library. For instance,

```
nvcc -c SortForC.cu
gcc Main.c S*o -L/usr/local/cuda/lib -lcudart
```

Here's an example:

```
// TestSort.cpp: interface to Thrust sort from non-CUDA callers

#include <stdio.h>

extern "C" void tsort(int *x, int *nx);

// test
int main()
{ int x[5] = {12,13,5,8,88};
  int n=5,*nx; nx = &n;
  int i;
  tsort(x,nx);
  for (i = 0; i < 5; i++) printf("%d\n",x[i]);
}
```

Compile and run for the OpenMP case:

```
g++ -g -mcmodel=medium -fopenmp -lgomp \
  -DTHRUST_DEVICE_BACKEND=THRUST_DEVICE_BACKEND_OMP \
  -I/home/matloff/Thrust SortForC.cpp -o sort.o -c
g++ -g -mcmodel=medium TestSort.cpp sort.o -fopenmp -lgomp
% a.out
5
8
12
13
88
```

And for the CUDA case:

```
nvcc SortForC.cu -c -o sfc.o
gcc -g TestSort.c sfc.o -L/usr/local/cuda/lib -lcudart
% a.out
5
8
12
13
88
```

6.4 Example: Calculating Percentiles in an Array

One of the most useful types of Thrust operations is that provided by conditional functions. For instance, **copy_if()** acts as a filter, copying from an array only those elements that satisfy a predicate. In the example below, for instance, we can copy every third element of an array, or every eighth etc.

```
// illustration of copy_if()

// find every k-th element in given array, going from smallest to
// largest; k obtained from command line and fed into ismultk() functor

// these are the the ik/n * 100 percentiles, i = 1, 2, ...

#include <stdio.h>

#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <thrust/sequence.h>
#include <thrust/remove.h> // for copy_if() but not copy_if.h

// functor
struct ismultk {
   const int increm; // k in above comments
   // get k from call
   ismultk(int _increm): increm(_increm) {}
   __device__
   bool operator()(const int i)
   { return i != 0 && (i % increm) == 0;
   }
};

// test
int main(int argc, char **argv)
```

```
{ int x[15] = {6,12,5,13,3,5,4,5,8,88,1,11,9,22,168};
  int n=15;
  thrust::device_vector<int> dx(x,x+n);
  thrust::sort(dx.begin(),dx.end());
  thrust::device_vector<int> seq(n);
  thrust::sequence(seq.begin(),seq.end(),0);
  thrust::device_vector<int> out(n);
  int incr = atoi(argv[1]); // k
  // for each i in seq, call ismultk() on this i, and if get a true
  // result, put dx[i] into out
  thrust::device_vector<int>::iterator newend =
    thrust::copy_if(dx.begin(),dx.end(),seq.begin(),out.begin(),
      ismultk(incr));
  thrust::copy(out.begin(), newend,
    std::ostream_iterator<int>(std::cout, " "));
  std::cout << "\n";
}
```

The **sequence()** function simply generates an array consisting of 0,1,2,...,n-1.

Our functor here is a little more advanced than the one we saw earlier. It now has an argument, which is **incr** in the case of our call,

```
thrust::copy_if(dx.begin(),dx.end(),seq,out,ismultk(incr));
```

That is then used to set a member variable in the struct:

```
const int increm; // k in above comments
ismultk(int _increm): increm(_increm) {}
```

This is in a sense the *second*, though nonexplicit, argument to our calls to **ismultk()**. For example, in our call,

```
thrust::copy_if(hx.begin(),hx.end(),seq,out,ismultk(incr));
```

the function designated by **operator** within the **ismultk** struct will be called individually on each element in **hx**, each one playing role of **i** in

```
bool operator()(const int i)
{ return i != 0 && (i % increm) == 0;
}
```

Since this code references **increm**, the value **incr** in our call above is used as well. The variable **increm** acts as a "global" variable to all the actions of the operator.

6.5 Mixing Thrust and CUDA Code

In order to mix Thrust and CUDA code, Thrust has the function **thrust::raw_pointer_cast()** to convert from a Thrust device pointer type to a CUDA device pointer type, and has **thrust::device_ptr** to convert in the other direction.

In our example in Section 6.6, we convert from Thrust to an ordinary address on the device:

```
int *wd;
...
wd = thrust::raw_pointer_cast(&w[0]);
...
{ if (i != 0 && (i % increm) == 0) wd[i] = 2 * wd[i];
```

In the other direction, say we start with a CUDA pointer, and want to use it in Thrust. We might have something like

```
int *dz;
...
cudaMalloc(&dz,100*sizeof(int));
...
thrust::device_ptr<int> tz(dz);
...
int k = thrust::reduce(tz,tz+100, (int) 0, thrust::plus<int>());
```

6.6 Example: Doubling Every kth Element of an Array

Let's adapt the code from the last section in order to illustrate another technique.

Suppose instead of copying every kth element of an array (after this first one), we wish to merely double each such element. There are various ways we could do this, but here we'll use an approach that shows another way we can use functors.

```
// double every k-th element in given array; k obtained from command
// line

#include <stdio.h>

#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/remove.h> // for copy_if()

// functor
struct ismultk
{
   const int increm; // k in above comments
```

```
14    const thrust::device_vector<int>::iterator w; // "pointer" to our array
15    int *wd;
16    // get "pointer," k
17    ismultk(thrust::device_vector<int>::iterator _w, int _increm):
18       w(_w), increm(_increm) {
19         wd = thrust::raw_pointer_cast(&w[0]);
20       }
21    __device__
22    bool operator()(const int i) // bool is phony, but void doesn't work
23    { if (i != 0 && (i % increm) == 0) wd[i] = 2 * wd[i];
24    }
25 };
26
27 // test
28 int main(int argc, char **argv)
29 { // test case:
30    int x[15] = {6,12,5,13,3,5,4,5,8,88,1,11,9,22,168};
31    int n=15;
32    thrust::device_vector<int> dx(x,x+n);
33    thrust::device_vector<int> seq(n);
34    thrust::sequence(seq.begin(),seq.end(),0);
35    thrust::device_vector<int> out(n);
36    int incr = atoi(argv[1]); // k
37    // for each i in seq, call ismultk() on this i, and if get a true
38    // result, put 0 in dx[i]
39    thrust::copy_if(dx.begin(),dx.end(),seq.begin(),out.begin(),
40       ismultk(dx.begin(),incr));
41    // did it work?
42    thrust::copy(dx.begin(), dx.end(),
43       std::ostream_iterator<int>(std::cout, " "));
44    std::cout << "\n";
45 }
```

Our functor here is quite different from before.

First, one of the functor's arguments is an iterator, rather than a simple type like **int**. This is really just like passing an array pointer to an ordinary C function.

Second, we've converted that iterator to a simple device array

```
1 const thrust::device_vector<int>::iterator w; // "pointer" to our array
2 int *wd;
3 ...
4     wd = thrust::raw_pointer_cast(&w[0]);
```

Our call to **copy_if()** doesn't actually do any copying. We are exploiting the "if" in "copy if," not the "copy."

6.7 Scatter and Gather Operations

These basically act as permuters; see the comments in the following small examples.

scatter:

```
// illustration of thrust::scatter(); permutes an array according to a
// map array

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/scatter.h>

int main()
{ int x[5] = {12,13,5,8,88};
   int n=5;
   thrust::device_vector<int> dx(x,x+n);
   // allocate map vector
   thrust::device_vector<int> dm(n);
   // allocate vector for output of gather
   thrust::device_vector<int> hdst(n);
   // example map
   int m[5] = {3,2,4,1,0};
   thrust::copy(m,m+n,dm.begin());
   thrust::scatter(dx.begin(),dx.end(),dm.begin(),ddst.begin());
   // the original x[0] should now be at position 3, the original x[1]
   // now at position 2, etc., i.e. 88,8,13,12;,5 check it:
   thrust::copy(ddst.begin(), ddst.end(),
      std::ostream_iterator<int>(std::cout, " "));
   std::cout << "\n";
}
```

gather():

```
// illustrations of thrust::gather(); permutes an array according to a
// map array

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/gather.h>

int main()
```

```
{ int x[5] = {12,13,5,8,88};
  int n=5;
  thrust::device_vector<int> dx(x,x+n);
  // allocate map vector
  thrust::device_vector<int> dm(n);
  // allocate vector for output of gather
  thrust::device_vector<int> ddst(n);
  // example map
  int m[5] = {3,2,4,1,0};
  thrust::copy(m,m+n,dm.begin());
  thrust::gather(dm.begin(),dm.end(),dx.begin(),ddst.begin());
  // the original x[3] should now be at position 0, the original x[2]
  // now at position 1, etc., i.e. 8,5,88,13,12; check it:
  thrust::copy(ddst.begin(), ddst.end(),
     std::ostream_iterator<int>(std::cout, " "));
  std::cout << "\n";
}
```

6.7.1 Example: Matrix Transpose

Here's an example of **scatter()**, applying it to transpose a matrix:

```
// matrix transpose, using scatter()

// similar to (though less efficient than) code included in the examples
// in the Thrust package

// matrices assumed stored in one dimension, row-major order

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/scatter.h>
#include <thrust/sequence.h>

struct transidx {
  const int nr; // number of rows in input
  const int nc; // number of columns in input
  // set nr, nc
  __host__ __device__
  transidx(int _nr, int _nc): nr(_nr), nc(_nc) {};
  // element i in input should map to which element in output?
  __host__ __device__
  int operator()(const int i)
```

```
22    { int r = i / nc; int c = i % nc; // row r, col c in input
23      // that will be row c and col r in output, which has nr cols
24      return c * nr + r;
25    }
26 };
27
28 int main()
29 { int mat[6] = { // test data
30      5, 12, 13,
31      3, 4, 5};
32    int nrow=2,ncol=3,n=nrow*ncol;
33    thrust::device_vector<int> dmat(mat,mat+n);
34    // allocate map vector
35    thrust::device_vector<int> dmap(n);
36    // allocate vector for output of gather
37    thrust::device_vector<int> ddst(n);
38    // construct map; element r of input matrix goes to s of output
39    thrust::device_vector<int> seq(n);
40    thrust::sequence (seq.begin(),seq.end());
41    thrust::transform(seq.begin(),seq.end(),dmap.begin(),transidx(nrow,ncol));
42    thrust::scatter(dmat.begin(),dmat.end(),dmap.begin(),ddst.begin());
43    // ddst should now hold the transposed matrix, 5,3,12,4,13,5; check it:
44    thrust::copy(ddst.begin(), ddst.end(), std::ostream_iterator<int>(std::cout
          , " "));
45    std::cout << "\n";
46 }
```

The idea is to determine, for each index in the original matrix, the index for that element in the transposed matrix. Not much new here in terms of Thrust, just more complexity.

It should be mentioned that the performance of this algorithm with a GPU backend would likely be better if matrix tiling were used (Section 12.2).

6.8 Advanced ("Fancy") Iterators

Since each Thrust call invokes considerable overhead, Thrust offers some special iterators to reduce memory access time and memory space requirements. Here are a few:

- **Counting iterators:** These play the same role as `thrust::sequence()`, but without actually setting up an array, thus avoiding the memory issues.
- **Transform iterators:** If your code first calls **thrust:transform()** and then makes another Thrust call on the result, you can combine them, which the Thrust people call **fusion**.
- **Zip iterators:** These essentially "zip" together two arrays (picture two halves of a zipper lining up parallel to each other as you zip up a coat). This is often useful when one needs to retain information on the position of an element within its array.

- **Discard iterators:** Sometimes we call **transform()** but don't need its output. Discard iterators then act in a manner similar to **/dev/null**.

6.8.1 Example: Matrix Transpose Again

Let's re-do the example of Section 6.7.1, this time using fusion.

```
// matrices assumed stored in one dimension, row-major order

#include <stdio.h>
#include <thrust/device_vector.h>
#include <thrust/scatter.h>
#include <thrust/sequence.h>
#include <thrust/iterator/transform_iterator.h>

struct transidx : public thrust::unary_function<int,int>
{
   const int nr; // number of rows in input
   const int nc; // number of columns in input
   // set nr, nc
   __host__ __device__
   transidx(int _nr, int _nc): nr(_nr), nc(_nc) {};
   // element i in input should map to which element in output?
   __host__ __device__
   int operator()(int i)
   { int r = i / nc; int c = i % nc; // row r, col c in input
      // that will be row c and col r in output, which has nr cols
      return c * nr + r;
   }
};

int main()
{ int mat[6] = {
     5, 12, 13,
     3, 4, 5};
   int nrow=2,ncol=3,n=nrow*ncol;
   thrust::device_vector<int> dmat(mat,mat+n);
   // allocate map vector
   thrust::device_vector<int> dmap(n);
   // allocate vector for output of gather
   thrust::device_vector<int> ddst(n);
   // construct map; element r of input matrix goes to s of output
   thrust::device_vector<int> seq(n);
   thrust::sequence (seq.begin(),seq.end());
```

```
38    thrust::scatter(
39       dmat.begin(),dmat.end(),
40       thrust::make_transform_iterator(seq.begin(),transidx(nrow,ncol)),
41       ddst.begin());
42    thrust::copy(ddst.begin(), ddst.end(),
43       std::ostream_iterator<int>(std::cout, " "));
44    std::cout << "\n";
45 }
```

The key new code here is:

```
thrust::scatter(
   dmat.begin(),dmat.end(),
   thrust::make_transform_iterator(seq.begin(),transidx(nrow,ncol)),
   ddst.begin());
```

Fusion requires a special type of iterator, whose type is horrendous to write. So, Thrust provides the **make_transform_iterator()** function, which we call to produce the special iterator needed, and then put the result directly into the second phase of our fusion, in this case into **scatter()**.

Essentially our use of **make_transform_iterator()** is telling Thrust, "Don't apply **transidx()** to **seq** yet. Instead, perform that operation as you go along, and feed each result of **transidx()** directly into **scatter()**." That word *direct* is the salient one here; it means we save n memory reads and n memory writes.[④] Moreover, we save the overhead of the kernel call, if our backend is CUDA.

Note that we also had to be a little bit more elaborate with data typing issues, writing the first line of our struct declaration as

```
struct transidx : public thrust::unary_function<int,int>
```

It won't work without this!

It would be nice to be able to use a counting iterator in the above code, but apparently the compiler encounters problems with determining where the end of the counting sequence is. There is similar code in the examples directory that comes with Thrust, and that one uses **gather()** instead of **scatter()**. Since the former specifies a beginning and an end for the map array, counting interators work fine.

6.9 A Timing Comparison

Let's look at matrix transpose one more time. First, we'll use the method, shown in earlier sections, of passing a device vector iterator to a functor. For variety, let's use Thrust's **for_each()** function. The following will be known as Code 1:

```
// matrix transpose, for_each version

#include <stdio.h>
```

④We are still writing to temporary storage, but that will probably be in registers (since we don't create the entire map at once), thus fast to access.

```
4  #include <thrust/device_vector.h>
5
6  // functor; holds iterators for the input and output matrices, and each
7  // invocation of the function copies from one element from the former to
8  // the latter
9  struct copyelt2xp
10 {
11    int nrow;
12    int ncol;
13    const thrust::device_vector<int>::iterator m; // input matrix
14    const thrust::device_vector<int>::iterator mxp; // output matrix
15    int *m1,*mxp1;
16    copyelt2xp(thrust::device_vector<int>::iterator _m,
17           thrust::device_vector<int>::iterator _mxp,
18           int _nr, int _nc):
19       m(_m), mxp(_mxp), nrow(_nr), ncol(_nc) {
20          m1 = thrust::raw_pointer_cast(&m[0]);
21          mxp1 = thrust::raw_pointer_cast(&mxp[0]);
22       }
23    __device__
24    void operator()(const int i)
25    // copies the i-th element of the input matrix to the output matrix
26    { // elt i in input is row r, col c there
27       int r = i / ncol; int c = i % ncol;
28       // that elt will be row c and col r in output, which has nrow
29       // cols, so copy as follows
30       mxp1[c*nrow+r] = m1[r*ncol+c];
31    }
32 };
33
34 // transpose nr x nc inmat to outmat
35 void transp(int *inmat, int *outmat, int nr, int nc)
36 {
37    thrust::device_vector<int> dmat(inmat,inmat+nr*nc);
38    // make space for the transpose
39    thrust::device_vector<int> dxp(nr*nc);
40    thrust::counting_iterator<int> seqb(0);
41    thrust::counting_iterator<int> seqe = seqb + nr*nc;
42    // for each i in seq, copy the matrix elt to its spot in the
43    // transpose
44    thrust::for_each(seqb,seqe,
45       copyelt2xp(dmat.begin(),dxp.begin(),nr,nc));
46    thrust::copy(dxp.begin(),dxp.end(),outmat);
```

```
}

int rand16() // generate random integers mod 16
{ return rand() % 16; }

// test code: cmd line args are matrix size, then row, col of elt to be
// checked

int main(int argc, char **argv)
{ int nr = atoi(argv[1]); int nc = nr;
   int *mat = (int *) malloc(nr*nc*sizeof(int));
   int *matxp = (int *) malloc(nr*nc*sizeof(int));
   thrust::generate(mat,mat+nr*nc,rand16);
   int checkrow = atoi(argv[2]);
   int checkcol = atoi(argv[3]);
   printf("%d\n",mat[checkrow*nc+checkcol]);
   transp(mat,matxp,nr,nc);
   printf("%d\n",matxp[checkcol*nc+checkrow]);
}
```

The **for_each()** function does what the name implies: It calls a function/functor for each element in a sequence, doing so in a parallel manner. Note that this also obviates our earlier need to use a discard iterator.

For comparison, we'll use the matrix transpose code that is included in Thrust's **examples/** file, to be referred to as Code 2:

```
// matrix transpose, from the Thrust package examples

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/gather.h>
#include <thrust/scan.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/iterator/transform_iterator.h>
#include <iostream>
#include <iomanip>
#include <stdio.h>

// convert a linear index to a linear index in the transpose
struct transpose_index : public thrust::unary_function<size_t,size_t>
{
  size_t m, n;

```

```
19   __host__ __device__
20   transpose_index(size_t _m, size_t _n) : m(_m), n(_n) {}
21
22   __host__ __device__
23   size_t operator()(size_t linear_index)
24   {
25       size_t i = linear_index / n;
26       size_t j = linear_index % n;
27
28       return m * j + i;
29   }
30 };
31
32 // convert a linear index to a row index
33 struct row_index : public thrust::unary_function<size_t,size_t>
34 {
35   size_t n;
36
37   __host__ __device__
38   row_index(size_t _n) : n(_n) {}
39
40   __host__ __device__
41   size_t operator()(size_t i)
42   {
43       return i / n;
44   }
45 };
46
47 // transpose an M-by-N array
48 template <typename T>
49 void transpose(size_t m, size_t n, thrust::device_vector<T>& src, thrust::
       device_vector<T>& dst)
50 {
51   thrust::counting_iterator<size_t> indices(0);
52
53   thrust::gather
54     (thrust::make_transform_iterator(indices, transpose_index(n, m)),
55      thrust::make_transform_iterator(indices, transpose_index(n, m)) + dst.size
           (),
56      src.begin(),
57      dst.begin());
58 }
59
```

```
60 void transp(int *inmat, int *outmat, int nr, int nc)
61 {
62    thrust::device_vector<int> dmat(inmat,inmat+nr*nc);
63    // make space for the transpose
64    thrust::device_vector<int> dxp(nr*nc);
65    transpose(nr,nc,dmat,dxp);
66    thrust::copy(dxp.begin(),dxp.end(),outmat);
67 }
68
69 int rand16() // generate random integers mod 16
70 { return rand() % 16; }
71
72 // test code: cmd line args are matrix size, then row, col of elt to be
73 // checked
74 int main(int argc, char **argv)
75 { int nr = atoi(argv[1]); int nc = nr;
76    int *mat = (int *) malloc(nr*nc*sizeof(int));
77    int *matxp = (int *) malloc(nr*nc*sizeof(int));
78    thrust::generate(mat,mat+nr*nc,rand16);
79    int checkrow = atoi(argv[2]);
80    int checkcol = atoi(argv[3]);
81    printf("%d\n",mat[checkrow*nc+checkcol]);
82    transp(mat,matxp,nr,nc);
83    printf("%d\n",matxp[checkcol*nc+checkrow]);
84 }
```

This approach is more efficient than ours in Section 6.7.1, making use of **gather()** instead of **scatter()**. It also takes advantage of fusion etc.

Code 1 is a lot easier to program than Code 2, but is it efficient? It turns out, though, that—good news!–the simpler code, i.e. Code 1, is actually a little faster than Code 2 in the case of a CUDA backend, and a lot faster in the OpenMP case.

Here we ran on CUDA backends, on a 10000x10000 matrix:

device	Code 1	Code 2
GeForce 9800 GTX	3.67	3.75
Tesla C2050	3.43	3.50

What about OpenMP? Here are some timing runs on a multicore machine (many more cores than the 16 we tried), using an input matrix of size 6000x6000:

# threads	Code 1	Code 2
2	9.57	23.01
4	5.17	10.62
8	3.01	7.42
16	1.99	3.35

6.10 Example: Transforming an Adjacency Matrix

Here is a Thrust approach to the example of Sections 4.13 and 5.12. To review, here is the problem:

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{6.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{6.2}$$

For instance, there is a 1 on the far right, second row of the above matrix, meaning that in the graph there is an edge from vertex 1 to vertex 3. This results in the row (1,3) in the transformed matrix seen above.

Here's Thrust code to do this:

```
1  // transgraph problem, using Thrust
2
3  #include <stdio.h>
4
5  #include <thrust/device_vector.h>
6  #include <thrust/transform.h>
7  #include <thrust/remove.h>
8  #include <thrust/iterator/discard_iterator.h>
9
10 // forms one row of the output matrix
11 struct makerow {
12   const thrust::device_vector<int>::iterator outmat;
13   int *om;
14   const int nc; // number of columns
15   makerow(thrust::device_vector<int>::iterator _outmat,int _nc) :
16      outmat(_outmat), nc(_nc) { om = thrust::raw_pointer_cast(&outmat[0]); }
17   __device__
```

```
18    // the j-th 1 is in position i of the orig matrix
19    bool operator()(const int i, const int j)
20    { om[2*j] = i / nc;
21       om[2*j+1] = i % nc;
22    }
23 };
24
25 int main(int argc, char **argv)
26 { int x[12] = {
27      0,1,1,0,
28      1,0,0,1,
29      1,1,0,0};
30    int nr=3,nc=4,nrc = nr*nc,i;
31    thrust::device_vector<int> dx(x,x+nrc);
32    thrust::device_vector<int> ones(x,x+nrc);
33    thrust::counting_iterator<int> seqb(0);
34    thrust::counting_iterator<int> seqe = seqb + nrc;
35    // get 1-D indices of the 1s
36    thrust::device_vector<int>::iterator newend =
37       thrust::copy_if(seqb,seqe,dx.begin(),ones.begin(),
38          thrust::identity<int>());
39    int n1s = newend - ones.begin();
40    thrust::device_vector<int> newmat(2*n1s);
41    thrust::device_vector<int> out(n1s);
42    thrust::counting_iterator<int> seq2b(0);
43    thrust::transform(ones.begin(),newend,seq2b,
44       thrust::make_discard_iterator(), makerow(newmat.begin(),nc));
45    thrust::copy(newmat.begin(), newmat.end(),
46       std::ostream_iterator<int>(std::cout, " "));
47    std::cout << "\n";
48 }
```

One new feature here is the use of counting iterators. First, we create two of them in the code

```
1 thrust::counting_iterator<int> seqb(0);
2 thrust::counting_iterator<int> seqe = seqb + nrc;
```

Here **seqb** (virtually) points to the 0 in 0,1,2,... Actually no array is set up, but references to **seqb** will act as if there is an array there. The counting iterator **seqb** starts at **nrc**, but its role here is simply to demarcate the end of the (virtual) array.

Now, how does the code work? The call to **copy_if()** has the goal of indentifying where in **dx** the 1s are located. This is accomplished by calling Thrust's **identity()** function, which just does f(x) = x, which is enough, as it will return either 1 or 0, the latter interpreted as True. In other words, the values between **seqb** and **seqe** will be copied whenever the corresponding values

in **dx** are 1s. The copied values are then placed into our array **ones**, which will now tell us where in **dx** the 1s are. Each such value, recall, will correspond to one row of our output matrix. The construction of the latter action is done by calling **transform()**:

```
thrust::transform(ones.begin(),newend,seq2b,
   thrust::make_discard_iterator(), makerow(newmat.begin(),nc));
```

The construction of the output matrix, **newmat**, is actually done as a side effect of calling **makerow()**. For this reason, we've set our third parameter to **thrust::make_discard_iterator()**. Since we never use the output from **transform()** itself, and it thus would be wasteful—of both memory space and memory bandwidth—to store that output in a real array. Hence we use a discard array instead.

Our algorithm consists of two stages—first finding the locations of the 1s, and then calculating the output matrix. Could we combine the two stages? Possibly, but there are difficulties to deal with.

The biggest problem is that we don't know the size of the output matrix in advance; counting the 1s separately gives us that information. Without that, we'd either have to make the output matrix too large initially and then shrink it, or continually expand it as we go through the computation. The latter would probably result in a major slowdown, as memory allocation takes time.

6.11 Prefix Scan

Thrust includes functions for prefix scan (see Chapter 11):

```
// illustration of parallel prefix sum

#include <stdio.h>

#include <thrust/device_vector.h>
#include <thrust/scan.h>

int main(int argc, char **argv)
{ int x[7] = {6,12,5,13,3,5,4};
   int n=7,i;
   thrust::device_vector<int> hx(x,x+n);
   // in-place scan; default operation is +
   thrust::inclusive_scan(hx.begin(),hx.end(),hx.begin());
   thrust::copy(hx.begin(), hx.end(),
      std::ostream_iterator<int>(std::cout, " "));
   std::cout << "\n";
}
```

6.12 More on Use of Thrust for a CUDA Backend

6.12.1 Synchronicity

Thrust calls are in fact CUDA kernel calls, and thus entail some latency. Other than the **transform()**-family functions, the calls are all synchronous.

6.13 Error Messages

A message like

```
terminate called after throwing an instance of 'std::bad_alloc'
  what(): std::bad_alloc
```

may mean that Thrust wasn't able to allocate your large array on the GPU.

Also, beware of the following. Consider the code

```
thrust::device_vector<int> seq(n);
thrust::copy_if(hx.begin(),hx.end(),seq,out,ismultk(hx.begin(),incr));
```

We forgot the **.begin()** for **seq**! If **seq** had been a non-Thrust array, declared as

```
int seq[n];
```

it would have been fine, but not for a Thrust array.

Unfortunately, the compiler gives us a very long megillah as an error message, a highly uninformative one. Keep this in mind if you get a 30-line compiler error.

The same thing happens if we forget to state the proper "include" files.

6.14 Other Examples of Thrust Code in This Book

- An application of Thrust's prefix-scan functionality is presented in Section 11.5

第 7 章　Message Passing Systems

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

7.1 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.

7.2 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 hardward for connection to its "neighbor" processors.

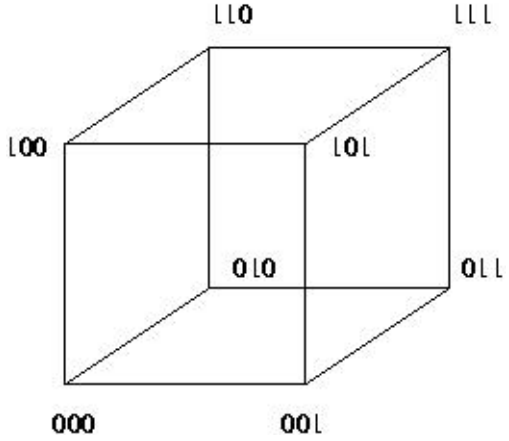
Hypercubes proved to be too expensive for the type of performance they could achieve, and the market was small anyway. Thus they are not common today, but they are still important, both for historical reasons (in the computer field, old techniques are often recycled decades later), and because the algorithms 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.

7.2.1 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.



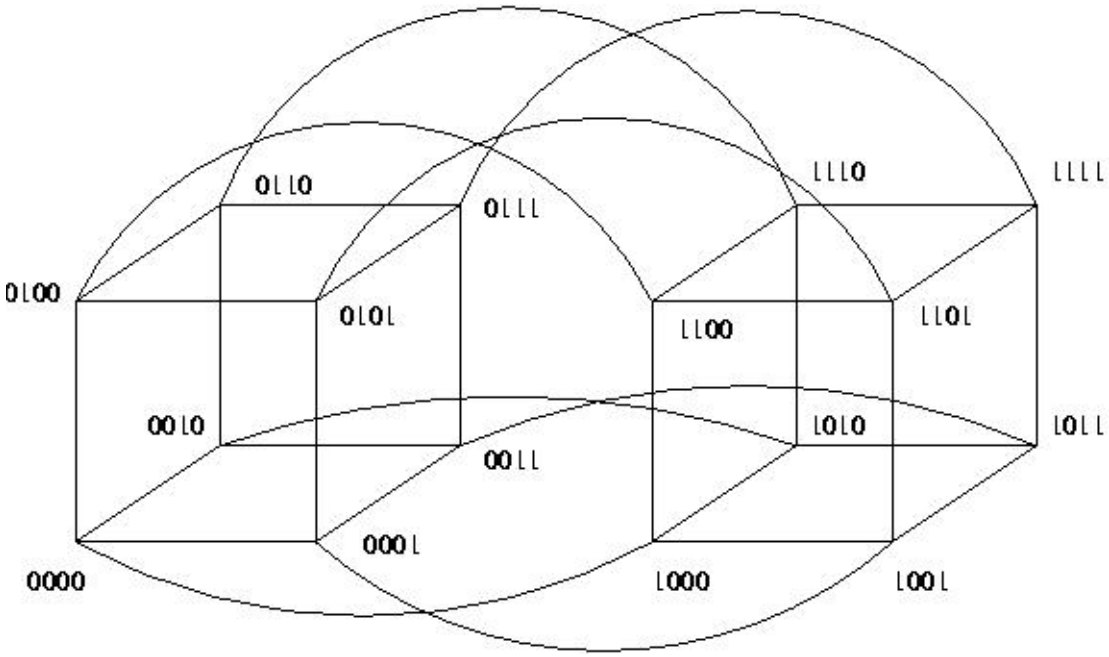
It is sometimes helpful to build up a cube from the lower-dimensional cases. To build a (d+1)-dimensional cube from two d-dimensional cubes, just follow this recipe:

(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:

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



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 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.

7.3 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,

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

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.

7.3.1 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.

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`.

7.3.2 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.

[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*.

7.4 Scatter/Gather Operations

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.

MPI includes scatter/gather operations in its wide offering of functions, and they are used in many MPI applications. R's **snow** package, which will be discussed in Section 10.5, is based entirely on scatter/gather, as is MapReduce, to be discussed below.

第 8 章　Introduction to MPI

MPI is the *de facto* standard for message-passing software.

8.1 Overview

8.1.1 History

Though (small) shared-memory machines have come down radically in price, to the point at which a dual-core PC is now commonplace in the home, historically shared-memory machines were available only to the "very rich"—large banks, national research labs and so on. This led to interest in message-passing machines.

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.

8.1.2 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.

8.1.3 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.

LAM is no longer being developed, and has been replaced by Open MPI (not to be confused with OpenMP). Personally, I still prefer the simplicity of LAM. It is still being maintained.

Note carefully: If your machine has more than one MPI implementation, make absolutely sure one is not interfering with the other. Make sure all execution and library paths all include one and only one implementation at a time.

8.1.4 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.

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.

Latency is quite different from **bandwidth**, which is the number of bits sent per second. Say the latency is 1.0 microsecond and the bandwidth is 1 gigabit, i.e. 1000000000 bits per second or 1000 bits per microsecond. Say the message is 2000 bits long. Then the first bit of the message arrives after 1 microsecond, and the last bit arrives after an additional 2 microseconds. In other words, the message is does not arrive fully at the destination until 3 microseconds after it is sent.

In the same setting, say bandwidth is 10 gigabits. Now the message would need 1.2 seconds to arrive fully, in spite of a 10-fold increase in bandwidth. So latency is a major problem even if the bandwidth is high.

For this reason, the MPI applications that run well on networks tend to be of the "embarrassingly parallel" type, with very little communication between the processes.

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 processes on that machine, the problem is less severe. In fact, some implementations of MPI communicate directly through shared memory in that case, rather than using the TCP/IP or other network protocol.

8.2 Review of Earlier Example

Though the presentation in this chapter is self-contained, you may wish to look first at the somewhat simpler example in Section 1.3.3, a pipelined prime number finder.

8.3 Example: Dijkstra Algorithm

8.3.1 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.

8.3.2 The MPI 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)
```

```
24
25  int nv,  // number of vertices
26      *notdone, // vertices not checked yet
27      nnodes,  // number of MPI nodes in the computation
28      chunk,  // number of vertices handled by each node
29      startv,endv,  // start, end vertices for this node
30      me,  // my node number
31      dbg;
32  unsigned largeint,  // max possible unsigned int
33           mymin[2],  // mymin[0] is min for my chunk,
34                      // mymin[1] is vertex which achieves that min
35           othermin[2],  // othermin[0] is min over the other chunks
36                         // (used by node 0 only)
37                         // othermin[1] is vertex which achieves that min
38           overallmin[2],  // overallmin[0] is current min over all nodes,
39                           // overallmin[1] is vertex which achieves that min
40           *ohd,  // 1-hop distances between vertices; "ohd[i][j]" is
41                  // ohd[i*nv+j]
42           *mind;  // min distances found so far
43
44  double T1,T2;  // start and finish times
45
46  void init(int ac, char **av)
47  {  int i,j,tmp; unsigned u;
48     nv = atoi(av[1]);
49     dbg = atoi(av[3]);
50     MPI_Init(&ac,&av);
51     MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
52     MPI_Comm_rank(MPI_COMM_WORLD,&me);
53     chunk = nv/nnodes;
54     startv = me * chunk;
55     endv = startv + chunk - 1;
56     u = -1;
57     largeint = u >> 1;
58     ohd = malloc(nv*nv*sizeof(int));
59     mind = malloc(nv*sizeof(int));
60     notdone = malloc(nv*sizeof(int));
61     // random graph
62     // note that this will be generated at all nodes; could generate just
63     // at node 0 and then send to others, but faster this way
64     srand(9999);
65     for (i = 0; i < nv; i++)
66        for (j = i; j < nv; j++)  {
67           if (j == i) ohd[i*nv+i] = 0;
68           else  {
69              ohd[nv*i+j] = rand() % 20;
70              ohd[nv*j+i] = ohd[nv*i+j];
71           }
72        }
73     for (i = 0; i < nv; i++)  {
74        notdone[i] = 1;
75        mind[i] = largeint;
76     }
77     mind[0] = 0;
78     while (dbg) ;  // stalling so can attach debugger
79  }
80
81  // finds closest to 0 among notdone, among startv through endv
82  void findmymin()
83  {  int i;
```

```
84    mymin[0] = largeint;
85    for (i = startv; i <= endv; i++)
86       if (notdone[i] && mind[i] < mymin[0])  {
87          mymin[0] = mind[i];
88          mymin[1] = i;
89       }
90 }
91
92 void findoverallmin()
93 {  int i;
94    MPI_Status status;  // describes result of MPI_Recv() call
95    // nodes other than 0 report their mins to node 0, which receives
96    // them and updates its value for the global min
97    if (me > 0)
98       MPI_Send(mymin,2,MPI_INT,0,MYMIN_MSG,MPI_COMM_WORLD);
99    else  {
100      // check my own first
101      overallmin[0] = mymin[0];
102      overallmin[1] = mymin[1];
103      // check the others
104      for (i = 1; i < nnodes; i++)  {
105         MPI_Recv(othermin,2,MPI_INT,i,MYMIN_MSG,MPI_COMM_WORLD,&status);
106         if (othermin[0] < overallmin[0])  {
107            overallmin[0] = othermin[0];
108            overallmin[1] = othermin[1];
109         }
110      }
111   }
112 }
113
114 void updatemymind()  // update my mind segment
115 {  // for each i in [startv,endv], ask whether a shorter path to i
116    // exists, through mv
117    int i, mv = overallmin[1];
118    unsigned md = overallmin[0];
119    for (i = startv; i <= endv; i++)
120       if (md + ohd[mv*nv+i] < mind[i])
121          mind[i] = md + ohd[mv*nv+i];
122 }
123
124 void disseminateoverallmin()
125 {  int i;
126    MPI_Status status;
127    if (me == 0)
128       for (i = 1; i < nnodes; i++)
129          MPI_Send(overallmin,2,MPI_INT,i,OVRLMIN_MSG,MPI_COMM_WORLD);
130    else
131       MPI_Recv(overallmin,2,MPI_INT,0,OVRLMIN_MSG,MPI_COMM_WORLD,&status);
132 }
133
134 void updateallmind()  // collects all the mind segments at node 0
135 {  int i;
136    MPI_Status status;
137    if (me > 0)
138       MPI_Send(mind+startv,chunk,MPI_INT,0,COLLECT_MSG,MPI_COMM_WORLD);
139    else
140       for (i = 1; i < nnodes; i++)
141          MPI_Recv(mind+i*chunk,chunk,MPI_INT,i,COLLECT_MSG,MPI_COMM_WORLD,
142             &status);
143 }
```

```
144
145 void printmind()  // partly for debugging (call from GDB)
146 {  int i;
147    printf("minimum distances:\n");
148    for (i = 1; i < nv; i++)
149       printf("%u\n",mind[i]);
150 }
151
152 void dowork()
153 {  int step,  // index for loop of nv steps
154        i;
155    if (me == 0) T1 = MPI_Wtime();
156    for (step = 0; step < nv; step++)  {
157       findmymin();
158       findoverallmin();
159       disseminateoverallmin();
160       // mark new vertex as done
161       notdone[overallmin[1]] = 0;
162       updatemymind(startv,endv);
163    }
164    updateallmind();
165    T2 = MPI_Wtime();
166 }
167
168 int main(int ac, char **av)
169 {  int i,j,print;
170    init(ac,av);
171    dowork();
172    print = atoi(av[2]);
173    if (print && me == 0)  {
174       printf("graph weights:\n");
175       for (i = 0; i < nv; i++)  {
176          for (j = 0; j < nv; j++)
177             printf("%u  ",ohd[nv*i+j]);
178          printf("\n");
179       }
180       printmind();
181    }
182    if (me == 0) printf("time at node 0: %f\n",(float)(T2-T1));
183    MPI_Finalize();
184 }
185
```

The various MPI functions will be explained in the next section.

8.3.3 Introduction to MPI APIs

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.

MPI_Comm_size() and MPI_Comm_rank()

In our function **init()** above, note the calls

```
1 MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
2 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**.

MPI_Send()

To see how MPI's basic send function works, consider our line above,

```
1 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

```
1 #define MYMIN_MSG 0
```

Receive calls, described in the next section, can ask to receive only messages of a certain type.

OMM_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**.

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 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**.

OMM_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.

8.4 Example: Removing 0s from an Array

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

#define MAX_N 100000
#define MAX_NPROCS 100
#define DATA_MSG 0
#define NEWDATA_MSG 1

int nnodes, // number of MPI processes
   n, // size of original array
   me, // my MPI ID
   has0s[MAX_N], // original data
   no0s[MAX_N], // 0-free data
   nno0s; // number of non-0 elements

int debug;
```

```
17
18 init(int argc, char **argv)
19 {
20    int i;
21    MPI_Init(&argc,&argv);
22    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
23    MPI_Comm_rank(MPI_COMM_WORLD,&me);
24    n = atoi(argv[1]);
25    if (me == 0) {
26       for (i = 0; i < n; i++)
27          has0s[i] = rand() % 4;
28    } else {
29       debug = atoi(argv[2]);
30       while (debug) ;
31    }
32 }
33
34 void managernode()
35 {
36    MPI_Status status;
37    int i;
38    int lenchunk;
39    lenchunk = n / (nnodes-1); // assumed divides evenly
40    for (i = 1; i < nnodes; i++) {
41       MPI_Send(has0s+(i-1)*lenchunk,lenchunk,
42          MPI_INT,i,DATA_MSG,MPI_COMM_WORLD);
43    }
44    int k = 0;
45    for (i = 1; i < nnodes; i++) {
46       MPI_Recv(no0s+k,MAX_N,
47          MPI_INT,i,NEWDATA_MSG,MPI_COMM_WORLD,&status);
48       MPI_Get_count(&status,MPI_INT,&lenchunk);
49       k += lenchunk;
50    }
51    nno0s = k;
52 }
53
54 void remov0s(int *oldx, int n, int *newx, int *nnewx)
55 { int i,count = 0;
56    for (i = 0; i < n; i++)
57       if (oldx[i] != 0) newx[count++] = oldx[i];
58    *nnewx = count;
59 }
```

```
60
61 void workernode()
62 {
63    int lenchunk;
64    MPI_Status status;
65    MPI_Recv(has0s,MAX_N,
66       MPI_INT,0,DATA_MSG,MPI_COMM_WORLD,&status);
67    MPI_Get_count(&status,MPI_INT,&lenchunk);
68    remov0s(has0s,lenchunk,no0s,&nno0s);
69    MPI_Send(no0s,nno0s,
70       MPI_INT,0,NEWDATA_MSG,MPI_COMM_WORLD);
71 }
72
73 int main(int argc,char **argv)
74 {
75    int i;
76    init(argc,argv);
77    if (me == 0 && n < 25) {
78       for (i = 0; i < n; i++) printf("%d ",has0s[i]);
79       printf("\n");
80    }
81    if (me == 0) managernode();
82    else workernode();
83    if (me == 0 && n < 25) {
84       for (i = 0; i < n; i++) printf("%d ",no0s[i]);
85       printf("\n");
86    }
87    MPI_Finalize();
88 }
```

8.5 Debugging MPI Code

If you are using GDB—either directly, or via an IDE such as Eclipse or Netbeans—the trick with MPI is to **attach** GDB to your running MPI processes.

Set up code like that we've seen in our examples here:

```
1 while (dbg) ;
```

This deliberately sets up an infinite loop of **dbg** is nonzero, for reasons to be discussed below.

For instance, suppose I'm running an MPI program **a.out**, on machines A, B and C. I would start the processes as usual, and have three terminal windows open. I'd log in to machine A, find the process number for **a.out**, using for example a command like **ps ax** on Unix-family systems, then attach GDB to that process. Say the process number is 88888. I'd attach by running the command

```
% gdb a.out 88888
```

That would start GDB, in the midst of my already-running process, thus stuck in the infinite loop seen above. I hit ctrl-c to interrupt it, which gives me the GDB prompt, `(gdb)`. I then type

```
(gdb) set var dbg = 0
```

which means when I next hit the **c** command in GDB, the program will proceed, not stuck in the loop anymore. But first I set my breakpoints.

8.6 Collective Communications

MPI features a number of **collective communication** capabilities, a number of which are used in the following refinement of our Dijkstra program:

8.6.1 Example: Refined Dijkstra Code

```
// 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.

8.6.2 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** to each node (including itself). 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.

Actually, this call to **MPI_Bcast()** is doing more than replacing the loop, since the latter had been part of an if-then-else that checked whether the given process had rank 0 or not.

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; N. Matloff, Analysis of a Programmed Backoff Method for Parallel Processing on Ethernets, in *Network-Based Parallel Computing*).

8.6.3 MPI_Reduce()/MPI_Allreduce()

Look at our call

```
1 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,

```
1 MPI_Reduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,0,MPI_COMM_WORLD);
2 MPI_Bcast(overallmin,1,MPI_2INT,0,MPI_COMM_WORLD);
```

could be replaced by

```
1 MPI_Allreduce(mymin,overallmin,1,MPI_2INT,MPI_MINLOC,MPI_COMM_WORLD);
```

Again, these can be optimized for particular platforms.

Here is a table of MPI reduce operations:

MPI_MAX	max
MPI_MIN	min
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	wordwise boolean and
MPI_LOR	wordwise boolean or
MPI_LXOR	wordwise exclusive or
MPI_BAND	bitwise boolean and
MPI_BOR	bitwise boolean or
MPI_BXOR	bitwise exclusive or
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

8.6.4 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

```
1 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 (including Node 0) contributes **chunk** number of MPI integers, from a location **mind+startv** in that node's program. Node 0 then receives **chunk** items sent from each node, stringing everything together in node order and depositing it all at **mind** in the program running at Node 0.

(Yes, the fifth argument is redundant with the second; same for the thrid and sixth.)

There is also **MPI_Allgather()**, which places the result at all nodes, not just one. Its call form is the same as **MPI_Gather()**, but with one fewer argument (since the identity of "the" gathering node is no longer meaningful):

```
1 int MPI_Allgather(srcbuf, srccount, srctype, destbuf, destcount, desttype,
     communicator)
```

8.6.5 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. For example, in the code in the next section, the call

```
1 MPI_Scatter(oh, lenchunk, MPI_INT, ohchunk, lenchunk, MPI_INT, 0,
2   MPI_COMM_WORLD);
```

means

> Node 0 will break up the array **oh** of type MPI_INT into chunks of length **lenchunk**, sending the i^{th} chunk to Node i, where **lenchunk** items will be deposited at **ohchunk**.

8.6.6 Example: Count the Number of Edges in a Directed Graph

Below is MPI code to count the number of edges in a directed graph. ("Directed" means that 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**.

```
lenchunk = nv / nnodes;
MPI_Scatter(oh, lenchunk, MPI_INT, ohchunk, lenchunk, 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);
```

8.6.7 Example: Cumulative Sums

Here we find cumulative sums. For instance, if the original array is (3,1,2,0,3,0,1,2), then it is changed to (3,4,6,6,9,9,10,12). (This topic is pursued in depth in Chapter 11.)

```
// finds cumulative sums in the array x

#include <mpi.h>
#include <stdlib.h>

#define MAX_N 10000000
#define MAX_NODES 10

int nnodes, // number of MPI processes
   n, // size of x
   me, // MPI rank of this node
   // full data for node 0, part for the rest
   x[MAX_N],
   csums[MAX_N], // cumulative sums for this node
   maxvals[MAX_NODES]; // the max values at the various nodes

int debug;
```

```
18
19 init(int argc, char **argv)
20 {
21    int i;
22    MPI_Init(&argc,&argv);
23    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
24    MPI_Comm_rank(MPI_COMM_WORLD,&me);
25    n = atoi(argv[1]);
26    // test data
27    if (me == 0) {
28       for (i = 0; i < n; i++)
29          x[i] = rand() % 32;
30    }
31    debug = atoi(argv[2]);
32    while (debug) ;
33 }
34
35 void cumulsums()
36 {
37    MPI_Status status;
38    int i,lenchunk,sum,node;
39    lenchunk = n / nnodes; // assumed to divide evenly
40    // note that node 0 will participate in the computation too
41    MPI_Scatter(x,lenchunk,MPI_INT,x,lenchunk,MPI_INT,
42       0,MPI_COMM_WORLD);
43    sum = 0;
44    for (i = 0; i < lenchunk; i++) {
45       csums[i] = sum + x[i];
46       sum += x[i];
47    }
48    MPI_Gather(&csums[lenchunk-1],1,MPI_INT,
49       maxvals,1,MPI_INT,0,MPI_COMM_WORLD);
50    MPI_Bcast(maxvals,nnodes,MPI_INT,0,MPI_COMM_WORLD);
51    if (me > 0) {
52       sum = 0;
53       for (node = 0; node < me; node++) {
54          sum += maxvals[node];
55       }
56       for (i = 0; i < lenchunk; i++)
57          csums[i] += sum;
58    }
59    MPI_Gather(csums,lenchunk,MPI_INT,csums,lenchunk,MPI_INT,
60       0,MPI_COMM_WORLD);
```

```
}

int main(int argc,char **argv)
{
   int i;
   init(argc,argv);
   if (me == 0 && n < 25) {
      for (i = 0; i < n; i++) printf("%d ",x[i]);
      printf("\n");
   }
   cumulsums();
   if (me == 0 && n < 25) {
      for (i = 0; i < n; i++) printf("%d ",csums[i]);
      printf("\n");
   }
   MPI_Finalize();
}
```

8.6.8 Example: an MPI Solution to the Mutual Outlinks Problem

Consider the example of Section 2.4.3. We have 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 MPI code below finds the mean number of mutual outlinks, among all pairs of vertices in a graph.

```
// MPI solution to the mutual outlinks problem

// adjacency matrix m is global at each node, broadcast from node 0

// assumes m is nxn, and number of nodes is < n

// for each node i, check all possible pairing nodes j > i; the various
// nodes work on values of i in a Round Robin fashion, with node k
// handling all i for which i mod nnodes = k

#include <mpi.h>
#include <stdlib.h>

#define MAXLENGTH 10000000

int nnodes, // number of MPI processes
   n, // size of x
```

```
18     me, // MPI rank of this node
19     m[MAXLENGTH], // adjacency matrix
20     grandtot; // grand total of all counts of mutuality
21
22  // get adjacency matrix, in this case just by simulation
23  void getm()
24  { int i;
25    for (i = 0; i < n*n; i++)
26       m[i] = rand() % 2;
27  }
28
29  init(int argc, char **argv)
30  {
31    int i;
32    MPI_Init(&argc,&argv);
33    MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
34    MPI_Comm_rank(MPI_COMM_WORLD,&me);
35    n = atoi(argv[1]);
36    if (me == 0) {
37       getm(); // get the data (app-specific)
38    }
39  }
40
41  void mutlinks()
42  {
43    int i,j,k,tot;
44    MPI_Bcast(m,n*n,MPI_INT,0,MPI_COMM_WORLD);
45    tot = 0;
46    for (i = me; i < n-1; i += nnodes) {
47       for (j = i+1; j < n; j++) {
48          for (k = 0; k < n; k++)
49             tot += m[twod2oned(n,i,k)] * m[twod2oned(n,j,k)];
50       }
51    }
52    MPI_Reduce(&tot,&grandtot,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
53  }
54
55  // convert 2-D subscript to 1-D
56  int twod2oned(n,i,j)
57  { return n * i + j; }
58
59  int main(int argc,char **argv)
60  { int i,j;
```

```
61    init(argc,argv);
62    if (me == 0 && n < 5) { // check test input
63       for (i = 0; i < n; i++) {
64          for (j = 0; j < n; j++) printf("%d ",m[twod2oned(n,i,j)]);
65          printf("\n");
66       }
67    }
68    mutlinks();
69    if (me == 0) printf("%f\n",((float) grandtot)/(n*(n-1)/2));
70    MPI_Finalize();
71 }
```

8.6.9 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.

8.6.10 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.

Here's how we might divide an MPI run into two groups (assumes an even number of MPI processes to begin with):

```
1  MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
2  MPI_Comm_rank(MPI_COMM_WORLD,&me);
3  ...
4  // declare variables to bind to groups
5  MPI_Group worldgroup, subgroup;
6  // declare variable to bind to a communicator
7  MPI_Comm subcomm;
8  ...
9  int i,startrank,nn2 = nnodes/2;
10 int *subranks = malloc(nn2*sizeof(int));
11 if (me < nn2) start = 0;
12 else start = nn2;
13 for (i = 0; i < nn2; i++)
14    subranks[i] = i + start;
15 // bind the world to a group variable
16 MPI_Comm_group(MPI_COMM_WORLD, &worldgroup);
17 // take worldgroup the nn2 ranks in "subranks" and form group
18 // "subgroup" from them
19 MPI_Group_incl(worldgroup, nn2, subranks, subgroup);
20 // create a communicator for that new group
```

```
21 MPI_Comm_create(MPI_COMM_WORLD, subgroup, subcomm);
22 // get my rank in this new group
23 MPI_Group_rank (subgroup, &subme);
```

You would then use **subcomm** instead of MPI_COMM_WORLD whenever you wish to, say, broadcast, only to that group.

8.7 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.

8.7.1 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**.

8.7.2 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.

8.7.3 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.

8.7.4 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

```
1 intMPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype,
2    int dest, int sendtag, int source, int recvtag, MPI_Comm comm,
3    MPI_Status *status)
```

Note that the sent and received messages can be of different lengths and can use different tags.

8.8 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 in Chapters 16 and 10.

8.9 Other MPI Examples in This Book

- The pipelined prime number finder in Chapter 1.
- Bucket sort with sampling, in Section 13.5.

第 9 章　Cloud Computing

In **cloud computing**, the idea is that a large corporation that has many computers could sell time on them, for example to make 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. Moreover, universities, businesses, research labs and so on are setting up their own small clouds, typically on clusters (a bunch of computers on the same local network, possibly with central controlling software for job management).

The paradigm that has become standard in the cloud today is MapReduce, developed by Google. In rough form, the approach is as follows. Various nodes server as *mappers*, and others serve as *reducers*.[①]

The terms *map* and **reduce** are in the functional programming sense. In the case of **reduce**, the idea is similar to reduction oeprations we've seen earlier in this book, such as the **reduction** clause in OpenMP and **MPI_Reduce()** for MPI. So, reducers in Hadoop perform operations such as summation, finding minima or maxima, and so on.

In this chapter we give a very brief introduction to Hadoop, today's open-source application of choice of MapReduce.

9.1 Platforms and Modes

In terms of platforms, Hadoop is basically a Linux product. Quoting from the Hadoop Quick Start, `http://hadoop.apache.org/common/docs/r0.20.2/quickstart.html#Supported+Platforms`:

> Supported Platforms:
>
> - GNU/Linux is supported as a development and production platform. Hadoop has been demonstrated on GNU/Linux clusters with 2000 nodes.
> - Win32 is supported as a *development platform*. Distributed operation has not been well tested on Win32, so it is not supported as a *production platform*.

Hadoop runs in one of three modes, of varying degrees of parallelism:

- standalone mode: Single mapper, single reducer, mainly useful for testing.
- pseudo-distributed mode: Single node, but multiple mapper and reducer threads.

[①] Of course, some or all of these might be threads on the same machine.

- fully-distributed mode: Multiple nodes, multiple mappers and reducers.

9.2 Overview of Operations

Here is a sumary of how a Hadoop application runs:

- Divide the input data into chunks of records. (In many cases, this is already the case, as a very large data file might be distributed across many machines.)
- Send the chunks to the mappers.
- Each mapper does some transformation (a "map," in functional programming terms) to each record in its chunks, and sends the output records to Hadoop.
- Hadoop collects the transformed records, splits them into chunks, sorts them, and then sends the chunks to the reducers.
- Each reducer does some summary operation (functional programming "reduce"), producing output records.
- Hadoop collects all those output records and concatenates them, producing the final output.

9.3 Role of Keys

The sorting operation, called the **shuffle** phase, is based on a **key** defined by the programmer. The key defines groups. If for instance we wish to find the total number of men and women in a certain debate, the key would be gender. The reducer would do addition, in this case adding 1s, one 1 for each person, but keeping a separate sum for each gender.

During the shuffle stage, Hadoop sends all records for a given key, e.g. all men, to one reducer. In other words, records for a given key will never be split across multiple reducers. (Keep in mind, though, that typically a reducer will have the records for many keys.)

9.4 Hadoop Streaming

Actually Hadoop is really written for Java or C++ applications. However, Hadoop can work with programs in any language under Hadoop's Streaming option, by reading from STDIN and writing to STDOUT, in text, line-oriented form in both cases. In other words, any executable program, be it Java, C/C++, Python, R, shell scripts or whatever, can run in Hadoop in streaming mode.

Everything is text-file based. Mappers input lines of text, and output lines of text. Reducers input lines of text, and output lines of text. The final output is lines of text.

Streaming mode may be less efficient, but it is simple to develop programs in it, and efficient enough in many applications. Here we present streaming mode.

So, STDIN and STDOUT are key in this mode, and as mentioned earlier, input and output are done in terms of lines of text. One additional requirement, though, is that the line format for both mappers and reducers must be

```
1 key \t value
```

where \t is the TAB character.

The usage of text format does cause some slowdown in numeric programs, for the conversion of strings to numbers and vice versa, but again, Hadoop is not designed for maximum efficiency.

9.5 Example: Word Count

The typical introductory example is word count in a group of text files. One wishes to determine what words are in the files, and how many times each word appears. Let's simplify that a bit, so that we simply want a count of the number of words in the files, not an individual count for each word.

The initial input is the lines of the files (combined internally by Hadoop into one superfile). The mapper program breaks a line into words, and emits (key,value) pairs in the form of (0,1). Our key here, 0, is arbitrary and meaningless, but we need to have one.

In the reducer stage, all those (key,value) pairs get sorted by the Hadoop internals (which has no effect in this case), and then fed into the reducers. Since there is only one key, 0, only one reducer will actually be involved. The latter adds up all its input values, i.e. all the 1s, yielding a grand total number of words in all the files.

Here's the pseudocode:

mapper:

```
for each line in STDIN
   break line into words, placed in wordarray
   for each word in wordarray
      # we have found 1 word
      print '0', '1' to STDOUT
```

reducer:

```
count = 0
for each line in STDIN
   split line into (key,value) # i.e. (0,1) here
   count += value # i.e. add 1 to count
print count
```

In terms of the key 0, the final output tells us how many words there were of type 0. Since we arbitrarily considered all words to be of type 0, the final output is simply an overall word count.

9.6 Example: Maximum Air Temperature by Year

A common Hadoop example on the Web involves data with the format for each line

```
year month day high_temperature air_quality
```

It is desired to find the maximum air temperature for each year.

The code would be very similar to the outline in the word count example above, but now we have a real key—the year. So, in the end, the final output will be a listing of maximum temperature by year.

Our mapper here will simply be a text processor, extracting year and temperature from each line of text. (This is a common paradigm for Hadoop applications.) The reducer will then do the maximum-finding work.

Here is the pseudocode:

mapper:

```
for each line in STDIN
  extract year and temperature
  print year, temperature to STDOUT
```

We have to be a bit more careful in the case of the reducer. Remember, though no year will be split across reducers, each reducer will likely receive the data for more than one year. It needs to find and output the maximum temperature for each of those years.[②]

Since Hadoop internals sort the output of the mappers by key, our reducer code can expect a bunch of records for one year, then a bunch for another year and so on. So, as the reducer goes through its input line by line, it needs to detect when one bunch ends and the next begins. When such an event occurs, it outputs the max temp for the bunch that just ended.

Here is the pseudocode:

reducer:

```
currentyear = NULL
currentmax = "-infinity"
for each line in STDIN
   split line into year, temperature
   if year == currentyear: # still in the current bunch
      currentmax = max(currentmax,temperature)
   else: # encountered a new bunch
      # print summary for previous bunch
      if currentyear not NULL:
         print currentyear, currentmax
      # start our bookkeeping for the new bunch
      currentyear = year
      currentmax = temperature
print currentyear, currentmax
```

9.7 Role of Disk Files

Hadoop has its own file system, HDFS, which is built on top of the native OS' file system of the machines.

[②] The results from the various reducers will then in turn be reduced, yielding the max temps for all years.

Very large files are possible, in some cases spanning more than one disk/machine. Indeed, this is the typical goal of Hadoop—to easily parallelize operations on a very large database. Files are typically gigabytes or terabytes in size. Moreover, there may be thousands of clusters, and millions of files.

This raises serious reliability issues. Thus HDFS is replicated, with each HDFS block existing in at least 3 copies, i.e. on at least 3 separate disks.

Disk files play a major role in Hadoop programs:

- Input is from a file in the HDFS system.
- The output of the mappers goes to temporary files in the native OS' file system.
- Final output is to a file in the HDFS system. As noted earlier, that file may be distributed across several disks/machines.

Note that by having the input and output files in HDFS, we minimize communications costs in shipping the data. The slogan used is "Moving computation is cheaper than moving data."

9.8 The Hadoop Shell

The HDFS can be accessed via a set of Unix-like commands. For example,

```
hadoop fs -mkdir somedir
```

creates a file **somedir** in my HDFS (invisible to me). Then

```
hadoop fs -put gy* somedir
```

copies all my files whose names begin with "gy" to **somedir**, and

```
hadoop fs -ls somedir
```

lists the file names in the directory **somedir**.

See `http://hadoop.apache.org/common/` for a list of available commands.

9.9 Running Hadoop

You run the above word count example something like this, say on the UCD CSIF machines.

Say my input data is in the directory **indata** on my HDFS, and I want to write the output to a new directory **outdata**. Say I've placed the mapper and reducer programs in my home directory (non-HDFS). I could then run

```
$ hadoop jar \
    /usr/local/hadoop-0.20.2/contrib/streaming/hadoop-0.20.2-streaming.jar \
     -input indata -output outdata \
     -mapper mapper.py -reducer reducer.py \
     -file /home/matloff/mapper.py \
     -file /home/matloff/reducer.py
```

This tells Hadoop to run a Java **.jar** file, which in our case here contains the code to run streaming-mode Hadoop, with the specified input and output data locations, and with the specified mapper and reducer functions. The **-file** flag indicates the locations of those functions (not needed if they are in my shell search path).

I could then run

```
hadoop fs -ls outdata
```

to see what files were produced, say **part_00000**, and then type

```
hadoop fs -cat outdata/part_00000
```

to view the results.

Note that the **.py** files must be executable, both in terms of file permissions and in terms of invoking the Python interpreter, the latter done by including

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

as the first line in the two Python files.

9.10 Example: Transforming an Adjacency Graph

Yet another rendition of the app in Section 4.13, but this time with a bit of problem, which will illustrate a limitation of Hadoop.

To review:

Say we have a graph with adjacency matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{9.1}$$

with row and column numbering starting at 0, not 1. We'd like to transform this to a two-column matrix that displays the links, in this case

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{9.2}$$

Suppose further that we require this listing to be in lexicographical order, sorted on source vertex and then on destination vertex.

At first, this seems right up Hadoop's alley. After all, Hadoop does sorting for us within groups

automatically, and we could set up one group per row of the matrix, in other words make the row number the key.

We will actually do this below, but there is a fundamental problem: Hadoop's simple elegance hinges on there being an independence between the lines in the input file. We should be able to process them one line at a time, independently of other lines.

The problem with this is that we will have many mappers, each reading only some rows of the adjacency matrix. Then for any given row, the mapper handling that row doesn't know what row number this row had in the original matrix. So we have no key!

The solution is to add a column to the matrix, containing the original row numbers. The matrix above, for instance, would become

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 2 & 0 & 1 & 0 & 1 \\ 3 & 1 & 1 & 1 & 0 \end{pmatrix} \tag{9.3}$$

Adding this column may be difficult to do, if the matrix is very large and already distributed over many machines. Assuming we do this, though, here is the mapper code (real code this time, not pseudocode):[③].

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

# map/reduce pair inputs a graph adjacency matrix and outputs a list of
# links; if say row 3, column 8 of the input is 1, then there will be a
# row (3,8) in the final output

import sys

for line in sys.stdin:
   tks = line.split() # get tokens
   srcnode = tks[0]
   links = tks[1:]
   for dstnode in range(len(links)):
      if links[dstnode] == '1':
          toprint = '%s\t%s' % (srcnode, dstnode)
          print toprint
```

And the reducer code:

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

import sys

for line in sys.stdin:
```

[③] There is a quick introduction to Python in Appendix C

```
    line = line.strip()
    print line # could remove the \t
```

Note that the row number, needed for other reasons, is also serving as our Hadoop key variable.

Recall that in the word count and yearly temperature examples above, the reducer did the main work, with the mappers playing only a supporting role. In this case here, it's the opposite, with the reducers doing little more than printing what they're given. *However*, keep in mind that Hadoop itself did a lot of the work, with its shuffle phase, which produced the sorting that we required in the output.

Here's the same code in R:

mapper:

```
#!/usr/bin/env Rscript

# map/reduce pair inputs a graph adjacency matrix and outputs a list of
# links; if say row 3, column 8 of the input is 1, then there will be a
# row (3,8) in the final output

con <- file("stdin", open = "r")
mapin <- readLines(con) # better not to read all at once, but keep simple
for (line in mapin) {
   tks <- strsplit(line,split=" ")
   tks <- tks[[1]]
   srcnode <- tks[1]
   links <- tks[-1]
   for (dstnode in 1:length(links)) {
      if (links[dstnode] == '1')
         cat(srcnode,"\t",dstnode,"\n",sep="")
   }
}
```

reducer:

```
#!/usr/bin/env Rscript

con <- file("stdin", open = "r")
mapin <- readLines(con)
for (line in mapin) {
   line <- strsplit(line,split="\t") # remove \t
   line <- line[[1]]
   cat(line,"\n")
}
```

9.11 Example: Identifying Outliers

In any large data set, there are various errors, say 3-year-olds who are listed as 7 feet tall. One way to try to track these down is to comb the data for outliers, which are data points (rows in the data set) that are far from the others. These may not be erroneous, but they are "suspicious," and we want to flag them for closer inspection.

In this simple version, we will define an outlier point to be one for which at least one of its variables is in the upper p proportion *in its group*. Say for example p is 0.02, and our groups are male adults and female adults, with our data variables being height and weight. Then if the height for some man were in the upper 2% of all men in the data set, we'd flag him as an outlier; we'd do the same for weight. Note that he'd be selected if either his height or his weight were in the top 2% for that variable among men in the data set. Of course, we could also look at the bottom 2%, or those whose Euclidean distance as vectors are in the most distant 2% from the centroid of the data in a group, etc.

We'll define groups in terms of combinations of variables. These might be, say, Asian male lawyers, female Kentucky natives registered as Democrats, etc. We'll use lexicographical order.

Say Variable 1 takes on the values 0-5, and Variable 2 has the values 0-12. The lex order would then be (0,0), (0,1),..., (0,11), (1,0), (1,1),...,(1,11),...,(5,0), (5,1),..., (5,11).

Note that the reducer reads in the entire data set first to determine where the upper percentiles are. This itself could be done with a separate MapReduce operation.

mapper:

```
#!/usr/bin/env Rscript

# map/reduce pair inputs a data matrix, forms groups according to
# combinations of specified integer variables, and then outputs the
# indices of outliers in each group

# any observation that has at least one variable in the top p% of its
# group is considered an outlier

# the first column of the input matrix is the observation number,
# starting at 0

# the group number for an observation is the linear index in
# lexicographical terms

# in Hadoop command line, use -mapper "olmap.R '0.4 2 3 6 16'" to
# specify parameters:

# p (given as a decimal number, e.g. 0.02); d, the number
# of data variables; g, the number of grouping variables; and finally g
# numbers which are the upper bounds for the last g-1 group variables
```

```
# (the lower bounds are always assumed to be 0)

init <- function() {
   ca <- commandArgs(trailingOnly=T)
   pars <- ca[1]
   pars <- strsplit(pars,split=" ")[[1]]
   pars <- pars[-1]
   ndv <<- as.integer(pars[1])
   ngv <<- as.integer(pars[2])
   # a few position variables, used in findgrp() below
   grpstart <<- 2
   grpend <<- grpstart + ngv - 1
   datastart <<- 2+ngv
   dataend <<- 1 + ngv + ndv
   # get upper bounds, and their reverse-cumulative products
   ubds <<- as.integer(pars[3:(1+ngv)])
   ubdsprod <<- vector(length=ngv-1)
   for (i in 1:(ngv-1))
      ubdsprod[i] <<- prod(ubds[i:(ngv-1)])
}

# converts vector of group variables to group number
findgrp <- function(grpvars) {
   sum <- 0
   for (i in 1:(ngv-1)) {
      m <- grpvars[i]
      sum <- sum + m * ubdsprod[i]
   }
   return(sum+grpvars[ngv])
}

# test
init()
con <- file("stdin", open = "r")
mapin <- readLines(con) # better not to read all at once
for (line in mapin) {
   tks <- strsplit(line,split=" ")
   tks <- tks[[1]]
   rownum <- tks[1]
   grpvars <- tks[grpstart:grpend]
   grpvars <- as.integer(grpvars)
   grpnum <- findgrp(grpvars)
   cat(grpnum,"\t",rownum," ",tks[datastart:dataend],"\n")
```

```
}
```

reducer:

```
#!/usr/bin/env Rscript

# see comments in olmap.R

# in Hadoop command line, use -reducer "olred.R '0.4 2 3 6 16'" or
# similar

# R's quantile() too complicated
quantl <- function(x,q) {
   return(sort(x)[ceiling(length(x)*q)])
}

init <- function() {
   ca <- commandArgs(trailingOnly=T)
   pars <- ca[1]
   pars <- strsplit(pars,split=" ")[[1]]
   # pars <- c("0.4","2","3","6","16") # for little test
   # pars <- c("0.1","2","2","3") # for big test
   p <<- as.double(pars[1])
   ndv <<- as.integer(pars[2])
}

emitoutliers <- function(datamat) {
   # find the upper p quantile for each variable (skip row number)
   toohigh <- apply(datamat[,-(1:2),drop=F],2,quantl,1-p)
   for (i in 1:nrow(datamat)) {
      if (any(datamat[i,-(1:2)] >= toohigh))
         cat(datamat[i,],"\n")
   }
}

# test
init()
con <- file("stdin", open = "r")
mapin <- readLines(con)
oldgrpnum <- -1
nmapin <- length(mapin)
for (i in 1:nmapin) {
   line <- mapin[i]
   tks <- strsplit(line,split="\t")[[1]]
```

```
41    grpnum <- as.integer(tks[1])
42    tmp <- strsplit(tks[2],split=" ")[[1]]
43    tmp <- tmp[tmp != ""] # delete empty element
44    row <- as.numeric(tmp)
45    if (oldgrpnum == -1) {
46       datamat <- matrix(c(grpnum,row),nrow=1)
47       oldgrpnum <- grpnum
48    } else if (grpnum == oldgrpnum) {
49       datamat <- rbind(datamat,c(grpnum,row))
50       if (i == nmapin) emitoutliers(datamat)
51    } else { # new group
52       emitoutliers(datamat)
53       datamat <- matrix(c(grpnum,row),nrow=1)
54       oldgrpnum <- grpnum
55    }
56 }
```

9.12 Debugging Hadoop Streaming Programs

One advantage of the streaming approach is that mapper and reducer programs can be debugged via normal tools, since those programs can be run on their own, without Hadoop, simply by using the Unix/Linux pipe capability.

This all depends on the fact that Hadoop essentially does the following Unix shell computation:

```
1 cat inputfile | mapperprog | sort -n | reducerprog
```

(Omit the **-n** if the key is a string.)

The mapper alone works like

```
1 cat inputfile | mapperprog
```

You thus can use whatever debugging tool you favor, to debug the mapper and reducer code separately.

Note, though, that the above pipe is *not quite* the same as Hadoop. the pipe doesn't break up the data, and there may be subtle problems arising as a result. But overall, the above approach provides a quick and easy first attempt at debugging.

The **userlogs** subdirectory of your Hadoop logs directory contains files that may be helpful, such as **stderr**.

9.13 It's a Lot More Than Just Programming

The real challenge in Hadoop is often not the programming, but rather the minimization of overhead. This involves things like tuning the file system, the number of mappers and reducers, and so on. These topics are beyond the scope of this book.

第 10 章　R 并行处理入门

10.1　为什么要在本书中用 R 语言?

在本书的其它章节里，C/C++ 依然是我们的主要语言，但我们也提供很多 R 语言的示例。为什么要用 R 呢?

- R 是最广泛使用的用于统计分析和数据处理的编程语言。在现今这个大数据时代，人民已经开发了相当数量的用于并行计算的 R 扩展包。特别地，**parallel** 扩展包现在已经是 R 基础包的一部分。
- R 语言的广泛使用，从 Google 设置了其内部的 R 语言规范一事就可见一斑[①]。现在 Oracle 也把 R 包含了自己的大数据分析方案中。
- 对于展示各种各样的并行算法，R 非常方便。这点的主要原因在于 R 内置了向量、矩阵和复数类型。

Python 也有很多并行库，比如 **multiprocessing**。关于 Python 的并行话题，我们会在第16章里讨论。

本章的示例会保持尽量简单。但 R 中的并行计算也可以应用到非常庞大而复杂的问题上。在附录B中，有一个 5 分钟的 R 快速入门。阅读时请牢记 R 中 list 结构。

R 中进行并行计算的关键就是——list 结构的操作。许多 R 的并行计算扩展包都非常依赖于 R 中的 list 结构。输入输出的参数和返回值经常都采用 list 的形式。读者可能有兴趣参考一下附录B中的相关内容。

10.2　R 和易并行问题（Embarrassing Parallel Problems）

需要注意的是，R 的并行扩展包一般只能处理易并行问题。正如在 2.3节中定义的，这些问题不仅容易并行化，而且信息传递的需求很少[②]。如我们所知，一般只有易并行问题会有很好的表现，但在 R 中情况尤其如此，原因如下。

R 语言的函数式编程的本质意味着，任何对一个向量或矩阵的元素的写入操作，比如

```
1 x[3] <- 8
```

都会重写整个向量或矩阵[③]。虽然有些例外（随着 R 版本更新，例外可能越来越多），但一般来说我们必须承认 R 中并行的向量和矩阵代码代价很高[④]。

对于不易并行的问题，大家应该考虑用 R 调用并行的 C 代码，这点会在10.9 节中讨论。

---

①个人角度来讲，我并不喜欢这些代码规范，我更喜欢我自己的。但从 Google 设置自己的 R 语言规范可以看出他们对 R 的重视程度。
②后面的要求把很多迭代算法排除在外了，尽管它们很容易并行化。
③R 中的元素赋值是一个函数调用，上面这个例子的参数分别为 `x`、3 和 8。
④R 中新的引用类（Reference class）可能会对此有所改变。

10.3 一些 R 的并行扩展包

这里我们列举了一些 R 的并行扩展包：

- Message-passing 或 scatter/gather (7.4节)：**Rmpi**、**snow**、**foreach**、**rmr**、**Rhipe**、**multicore**[5]、**rzmq**
- 内存共享：**Rdsm**、**bigmemory**
- GPU：**gputools**、**rgpu**

大家可以从 `http://cran.r-project.org/web/views/HighPerformanceComputing.html`找到更加详尽的列表。

从 2.14 版本开始，R 默认包括了由 **snow** 和 **multicore** 构成的 **parallel** 扩展包。（早期版本可能需要分别下载。）正是因为如此，二者都在范围之内。另外，我们也会讨论 **Rdsm/bigmemory** 和 **gputools**。

10.4 安装和载入这些扩展包

安装：

需要注意的是，如果你使用的是 2.14 版或更高版本的 R，你已经安装了 **snow** 和 **multicore**。一般来说，除了 **rgpu**，其它所有扩展包都可以从 R 官方的代码仓库 CRAN（`http://cran.r-project.org`）下载。这里以 **snow** 为例：

加入你想把它安装在**/a/b/c/** 目录下。最简单的方法就是使用 R 的函数：

```
> install.packages("snow","/a/b/c/")
```

这会将 **snow** 安装在**/a/b/c/snow** 目录下。

之后你需要将目录**/a/b/c**（不是**/a/b/c/snow**）加到你的 R 搜索路径中。我推荐大家在自己 home 目录下的**.Rprofile** 文件（这是 R 的启动设置文件）中添加这样一行。

```
.libPaths("/a/b/c/")
```

在一些情况下，由于所需库的位置原因，你可能需要手动安装一个 CRAN 上的扩展包。这一点请参考下面的10.8.1节和10.8.3节。

载入一个扩展包：

通过调用 **library()** 来载入一个扩展包。例如，载入 **parallel**，可以使用：

```
> library(parallel)
```

10.5 R 中的 snow 扩展包

snow 最大的优点在于其简单。其概念和实现都非常简单，能出错的地方不多。因此，它可能是现在使用最广泛的 R 并行包。

snow 扩展包可以直接通过 network socket 运行（由于用户只需要安装 **snow**，着可能是最常见的用法），也可以运行于 **Rmpi**（R 的 MPI 接口）、PVM 或 NWS 之上。

[5] **multicore** 扩展包运行于多核内存共享的平台之上，但在读写过程中并不共享数据。

它也可以在一个 scatter/gather 模型（7.4节）下进行操作。正如 R 中的 **apply()** 函数会将同样的函数作用于一个矩阵的每行上（见下面的示例），**snow** 中的 **parApply()** 会在多台机器上并行地完成类似的操作；不同的机器会操作不同的行。（除了使用多台机器，我们也可以在多核的机器上运行多个 **snow** client。）

10.5.1 使用

在使用

```
> library(snow)
```

载入 **snow** 之后，通过调用 **snow** 中的 **makeCluster()** 函数，我们可以设置一个 **snow** 集群。该函数的 **type** 参数用于选择网络平台，诸如 “MPI” 或 “SOCK”。后者用于将 **snow** 运行于其自己创建的 TCP/IP sockets 之上，而不是使用 MPI 。

在这个例子里，我在名为 **pc48** 和 **pc49** 的电脑上使用 “SOCK” 选择，以这种方式设置集群[6]：

```
> cls <- makeCluster(type="SOCK",c("pc48","pc49"))
```

需要注意的是上面的 R 代码在名为 **pc48** 和 **pc49** 的机器上设置了**工作节点**；这和**管理节点**相区别，管理节点运行于执行 R 代码的机器上。

如果你想把工作节点和管理节点同时运行在同一台机器上（特别是在一台多核的机器上），需要使用 **localhost** 作为机器名。

还有其它很多可选的参数。一个你可能觉得非常有用的是 **outfile**，它会把调用的结果记录在名为 **outfile** 的文件里。这在调用失败进行 debug 时非常有用。

10.5.2 示例：使用 parApply() 进行矩阵向量相乘

为了介绍 **snow**，让我们考虑一个简单的矩阵向量相乘的简单示例。我是指一个测试矩阵如下：

```
> a <- matrix(c(1:12),nrow=6)
> a
     [,1] [,2]
[1,] 1 7
[2,] 2 8
[3,] 3 9
[4,] 4 10
[5,] 5 11
[6,] 6 12
```

我们会将向量 $(1,1)^T$ （T 这里表示转置）和矩阵相乘。在这个简单的示例，我们当然可以直接完成：

```
> a %*% c(1,1)
     [,1]
[1,] 8
```

[6] 如果你使用的是一个文件共享系统的电脑集群，尽量保证 R 的安装路径一致，以避免问题。

```
[2,] 10
[3,] 12
[4,] 14
[5,] 16
[6,] 18
```

但是让我们看看如何使用 R 的 **apply()** 来完成它。尽管这仍是顺序执行，但这为我们扩展到并行计算提供了便利。

R 的 **apply()** 函数调用一个用户定义的标量函数作用于用户指定的矩阵的每一行（或每一列）。为了将 **apply()** 用于这里的矩阵向量相乘问题，我们定义一个点积的函数：

```
> dot <- function(x,y) {return(x%*%y)}
```

现在调用 **apply()**：

```
> apply(a,1,dot,c(1,1))
[1] 8 10 12 14 16 18
```

这个调用将函数 **dot()** 作用于矩阵 **a** 的每一行（这个可以从 1 看出，2 意味着每一列）；每一行都将作为 **dot()** 的第一个参数，而 c(1,1) 会作为第二个参数。换言之，**dot()** 的第一次调用就是

```
dot(c(1,7),c(1,1))
```

snow 中的 **parApply()** 函数将 **apply()** 扩展到并行计算。我们把它用于将我们的矩阵相乘问题并行化，运行在我们名为 **cls** 的集群之上：

```
> parApply(cls,a,1,dot,c(1,1))
[1] 8 10 12 14 16 18
```

parApply() 所作的就是将矩阵每一行发送给每一个节点，同时发送的还由函数 **dot()** 和参数 **c(1,1)**。每个节点将 **dot()** 作用到接收的行上，之后将结果返回给管理节点。

R 的 **apply()** 函数一般只用于变量值的情形，这意味着 **apply(m,i,f)** 调用中的函数 **f()** 的返回值是标量。如果 **f()** 的返回值是向量值，那返回的会是一个矩阵而不是一个向量，矩阵里的每一列是 **f()** 作用于 **m** 的一列或一行的结果。**parApply()** 也同样如此。

10.5.3 snow 中的其它函数：clusterApply()、clusterCall() 等

上一节，我们介绍了 **parApply()** 函数。它可以这样调用

- **parApply()**:

```
parApply(cls,m,DIM,f,...)}
```

这个调用会把矩阵 **m** 的每一行分配到 **cls** 的各个工作节点，之后函数 **f()** 会被作用到每一行，省略号在这里表示可选参数。参数 **DIM** 为 1 时表示行操作，2 表示列操作。

返回值是一个向量（也可能是个矩阵，如上所述）。

snow 最大的有点在于其简单，因此并没有很多复杂的函数，但当然不止 **parApply()** 一个。这里列举了一些：

- **clusterApply()**:

这个函数可能是 **snow** 中被使用最频繁的函数。

```
clusterApply(cls,individualargs,f,...)}
```

这会使 **f()** 在 **cls** 中的每个节点上运行。这里的 **individualargs** 是一个 R 列表（如果是个向量，会被转换成列表）。当 **f()** 在集群中的节点 i 上被调用时，其参数如下所述：第一个参数是 **individualargs** 的第 i 个元素，或者说是 **individualargs**[[i]]；如果在调用时，是用了省略号所代表的（可选）参数，它们会作为第二、第三或更多的参数传递给 **f()**。

如果 **individualargs** 的元素数量大于集群中的节点数，那么 **cls** 会被循环使用（可以把它作为一个向量对待），所以多数或全部节点会在不止一个 **individualargs** 元素上调用 **f()**。返回值是一个 R 列表，其中第 i 个元素是 **f()** 作用于 **individualargs** 中第 i 个元素的结果。所以说，**individualargs** 列表又需要拆分并行计算的工作构成。

- **clusterApplyLB():**

  这是 **clusterApply()** 的负载均衡模式，目的在于解决我们在第2章中提到的性能问题。

  为了解释 **clusterApply()** 的两者形式的区别，假设我们的集群由 10 个节点，而我们有 25 个需要执行的任务（或者说 **individualargs** 的长度是 25）。如果使用 **clusterApply()**，会发生下列这些：

  – 前 10 个任务会被分配给工作节点，每个节点一个任务。
  – 管理节点会等这 10 个任务完成，之后再分配另外 10 个。
  – 管理节点会等这 10 个任务完成，之后在分配剩下的 5 个。
  – 管理节点会等这 5 个任务完成，之后返回 25 个结果。

  而是用 **clusterApplyLB()** 时，会按照下面这种方式执行：

  – 前 10 个任务会被分配给工作节点，每个节点一个任务。
  – 当由节点任务结束时，管理节点会马上行动，将第 11 个任务分配给这个节点，即使其它节点的任务还没完成。
  – 管理节点会继续照此工作，一旦一个节点任务完成，就会分配新的任务，知道所有任务完成。
  – 管理节点最后会返回 25 个结果。

  用第2章和 OpenMP 一章中的4.3.3节的说法，**clusterApply()** 使用了一种**静态**的调度策略，而 **clusterApplyLB()** 使用了一种动态策略；其中 chunk size 为 1。

- **clusterCall():**

  函数 **clusterCall(cls,f,...)** 将函数 **f()** 和省略号所代表的参数（如果有的话），发送到每个工作节点。在每个节点上，**f()** 会使用这些参数求值。返回值是一个 R 列表，第 i 个元素师第 i 个节点的计算结果。（一眼看上去，似乎每个节点都会返回同样的结果，但 `f()` 会使用每个节点特定的参数，从而返回不同的结果。）

- **clusterExport():**

  函数 **clusterExport(cls,varlist)** 会将名字出现在字符向量 **varlist** 中的变量拷贝到 **cls** 中的各个节点。你可以使用这个函数来避免从管理节点到工作节点开销巨大的数据传输。使用这个函数，你可以只传输数据集一次；通过在相应的变量上使用 **clusterExport()**，之后在工作节点上将其作为全局变量使用。同样地，返回值仍是个 R 列表，第 i 个元素师集群中第 i 个节点的计算结果。

默认情况下，被传输到工作节点的变量在管理节点上必须是全局变量。

需要特别注意的是，一旦你传输了一个变量，比如 **x**，从管理节点到各个工作节点上，各个拷贝和工作节点上的变量就是独立的了（各个拷贝之间也是相互独立的）。如果其中一个拷贝改变了，在其他拷贝中不会反应这些变化。

- **clusterEvalQ()：**

  函数 **clusterEvalQ(cls,expression)** 会在 **cls** 的各个节点上运行 **expression**。

10.5.4 示例：并行求和

现在让我们再看一个示例，我们用 **snow** 来进行并行求和。先从一个很简单的版本开始，之后再考虑复杂的版本。

```
parsum <- function(cls,x) {
  # 在节点上分配 x 的索引（实际上没有传输任何东西）
  xparts <- clusterSplit(cls,x)
  # 现在传输到节点上，并进行求和
  tmp <- clusterApply(cls,xparts,sum)
  # 现在将各个单独的加和合并得到结果
  tot <- 0
  for (i in 1:length(tmp)) tot <- tot + tmp[[i]]
  return(tot)
}
```

现在我们在一个有两个共走节点的集群 **cls** 上进行测试：

```
> x
[1] 1 2 3 4 5 6 5 12 13
> parsum1(cls,x)
[1] 51
```

结果不错。现在我们来想一下，这是如何完成的？

最基本的想法就是讲我们的向量分块，之后分配给工作节点。每个工作节点会把所分配的小块求和，再把结果返回给管理节点。管理节点会把这些结果求和，返回我们想要的求和的最终结果。

为了将我们的向量 **x** 分块并发给各个节点，我先来看 **snow** 中的函数 **clusterSplit()**。这个函数的输入是一个 R 向量，之后将其分块，分块的数量和工作节点数相同。

例如，在上面的两个工作节点的集群上，我们得到：

```
> xparts <- clusterSplit(cls,x)
> xparts
[[1]]
[1] 1 2 3 4

[[2]]
[1] 5 6 5 12 13
```

非常肯定的是，我们的列表 **xparts** 有在其一个元素中有 **x** 的一块，而另一个元素中有 **x** 的另一块。之后这两块被传输到两个工作节点上：

```
> tmp <- clusterApply(cls,xparts,sum)
> tmp
[[1]]
[1] 10

[[2]]
[1] 41
```

同样像 **snow** 中的其他函数一样，**clusterApply()** 会以列表的形式返回结果。这里我们将结果赋值给了 **tmp**。其内容如下

```
> tmp
[[1]]
[1] 10

[[2]]
[1] 41
```

也就是 **x** 每一小块的和。

为了得到最后的结果，我们不能简单地在 **tmp** 上使用 R 中 **sum()** 函数：

```
> sum(tmp)
Error in sum(tmp) : invalid 'type' (list) of argument
```

这是因为 **sum()** 接受的是向量，而不是列表。所以我们自己写一个循环来把结果加起来：

```
tot <- 0
for (i in 1:length(tmp)) tot <- tot + tmp[[i]]
```

需要注意的一点是，我们使用 `[[]]`来获取列表中的元素。

我可以通过调用 R 中的 **Reduce()** 函数来取代上面的循环，从而对代码进行优化。**Reduce()** 很像4.3.5节和8.6.3节中的 reduction 操作。（注意，这里是个串行操作，不是并行。）一般以 **Reduce(f,y)** 这种形式使用，它对函数 **f()** 和列表 **y** 进行如下操作

```
z <- y[1]
for (i in 2:length(y)) z <- f(z,y[i])
```

使用 **Reduce()** 可以使代码更紧凑可读，一些情况下还会提高执行效率（我们这里只有很少的项目进行相加，暂时还不用考虑效率）。而且，**Reduce()** 会将 **tmp** 从一个列表转换为向量，这就解决了我们直接对 **tmp** 使用 **sum()** 时的问题。

下面是新的代码：

```
parsum <- function(cls,x) {
  xparts <- clusterSplit(cls,x)
  tmp <- clusterApply(cls,xparts,sum)
```

```
  Reduce(sum,tmp) # implicit return()
}
```

需要说明的是，在 R 中，如果没有显式的 **return()** 语句，那最后求得的值会被作为返回值，这里是 **Reduce()** 的计算结果。

Reduce() 是一个非常便利的函数，特别是在和 **snow** 一起使用时。这里有一个我们把多个矩阵进行合并的示例：

```
> Reduce(rbind,list(matrix(5:8,nrow=2),3:4,c(-1,1)))
     [,1] [,2]
[1,] 5 7
[2,] 6 8
[3,] 3 4
[4,] -1 1
```

rbind() 函数只有两个参数，在这里我们有三个。通过使用 **Reduce()** 可以解决这个问题。

10.5.5 示例：对角分块矩阵求逆

假设我们有一个对角分块矩阵，比如

$$\begin{pmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 8 & 1 \\ 0 & 0 & 1 & 5 \end{pmatrix}$$

我们想对其求逆。这是个易并行问题：假如我们有两个处理器，我们可以很简单地让其中之一对第一个 2×2 子矩阵求逆，让另一个对第二个 2×2 子矩阵求逆，之后我们将两个逆矩阵放回原来的位置。

通讯的开销在这里不是很大，一个 n×n 矩阵求逆的时间复杂度为 $O(n^3)$，而通讯只有 $O(n^2)$。

现在我们讨论一下用于分块对角矩阵求逆的 **snow** 代码。

```
# invert a block diagonal matrix m, whose sizes are given in szs;
# return value is the inverted matrix
bdiaginv <- function(cls,m,szs) {
  nb <- length(szs) # number of blocks
  dgs <- list() # will form args for clusterApply()
  rownums <- getrng(szs)
  for (i in 1:nb) {
     rng <- rownums[i,1]:rownums[i,2]
     dgs[[i]] <- m[rng,rng]
  }
  invs <- clusterApply(cls,dgs,solve)
  for (i in 1:nb) {
     rng <- rownums[i,1]:rownums[i,2]
     m[rng,rng] <- invs[[i]]
  }
```

```
  m
}

# find row number ranges for the blocks, returned in a # 2-column
# matrix; blkszs = block sizes
getrng <- function(blkszs) {
  col2 <- cumsum(blkszs) # cumulative sums function
  col1 <- col2 - (blkszs-1)
  cbind(col1,col2) # column bind
}
```

我们来检测一下：

```
> m
     [,1] [,2] [,3] [,4] [,5]
[1,] 1 2 0 0 0
[2,] 7 8 0 0 0
[3,] 0 0 1 2 3
[4,] 0 0 2 4 5
[5,] 0 0 1 1 1
> bdiaginv(cls,m,c(2,3))
          [,1] [,2] [,3] [,4] [,5]
[1,] -1.333333 0.3333333 0 0 0
[2,] 1.166667 -0.1666667 0 0 0
[3,] 0.000000 0.0000000 1 -1 2
[4,] 0.000000 0.0000000 -3 2 -1
[5,] 0.000000 0.0000000 2 -1 0
```

这里的 **szs** 参数，包含了分块的大小。由于我们只有一个 2×2 和一个 3×3 的块，分块的大小就是 2 和 3，因为在函数调用里使用 **c(2,3)**。

这里 **clusterApply()** 的使用和早先的例子很相似。代码中值得注意的地方是我们需要保存每一块在大矩阵中的位置。最后我们写了一个 **getrng()** 函数，用于返回不同块的起始和结束的行数。我们通过使用这个函数来设置 **clusterApply()** 的 **dg** 参数：

```
for (i in 1:nb) {
  rng <- rownums[i,1]:rownums[i,2]
  dgs[[i]] <- m[rng,rng]
```

大家要记得表达式 **m[rng,rng]** 会提取 **m** 的行和列出来，在这里就是第 i 块。

10.5.6 示例：Mutual Outlink

让我们考虑2.4.3节中的例子。我们有一个网络，比如 web 链接。对于其中的两个节点，比如两个网站，我可能对其 mutual outlink 感兴趣，也就是两个网站共同的对外链接。

下面的 **snow** 代码会计算整个网络中任意一对节点 mutual outlink 的均值。

```
# snow version of mutual links problem

library(snow)

mtl <- function(ichunks,m) {
   n <- ncol(m)
   matches <- 0
   for (i in ichunks) {
      if (i < n) {
         rowi <- m[i,]
         matches <- matches +
            sum(m[(i+1):n,] %*% as.vector(rowi))
      }
   }
   matches
}

# returns the mean number of mutual outlinks in m, computing on the
# cluster cls
mutlinks <- function(cls,m) {
   n <- nrow(m)
   nc <- length(cls)
   # determine which worker gets which chunk of i
   options(warn=-1)
   ichunks <- split(1:n,1:nc)
   options(warn=0)
   counts <- clusterApply(cls,ichunks,mtl,m)
   do.call(sum,counts) / (n*(n-1)/2)
}
```

对于 **m** 中的每一行，我们会计算其下面每一行中的 mutual link。为了在工作节点之间分配工作，我们可以如下使用 **clusterSplit()**

```
clusterSplit(cls,1:nrow(m))
```

但这会有一个在2.4.3节中讨论过的不均衡问题。比如我们有两个工作节点和 100 行。如果我们像上一节一样使用 **clusterSplit()**，第一个节点进行的比较工作会远比第二个节点多。

一个解决方案是在调用 **clusterSplit()** 之前，将行号随机打乱。另一方法，也是我们上面的代码中使用的，就是用 R 的 **split()** 函数。

那 **split()** 是做什么的？它根据第二个参数中设置的 “类别”，将第一个参数进行分块处理。我们来看这个示例：

```
> split(2:5,c('a','b'))
$a
[1] 2 4
```

```
4
5 $b
6 [1] 3 5
```

这里的种类是 a 和 b。**split()** 函数要求第二个参数和第一个参数长度相同，所以首先会对第二个参数进行 “循环” 处理成 a,b,a,b,a。之后会将 2 和 4 放入类别 a，将 3 和 5 放入类别 b。**split()** 函数最后会返回一个相应的列表。

现在我们再回到上面的 **snow** 示例，我们仍然假设在两个工作节点中分配 100×100 的矩阵 **m**，代码

```
1 nc <- length(cls)
2 ichunks <- split(1:n,1:nc)
```

会生成一个由两部分构成的列表，第一部分由奇数行构成，第二部分由偶数行构成。之后我们再使用

```
1 counts <- clusterApply(cls,ichunks,mtl,m)
```

就可以在两个工作节点间实现一个负载均衡了。

注意这个调用需要将 **m** 作为一个参数（作为函数 **mtl()** 的参数）。否则工作节点将没有可供使用的 **m**。另一个选择是使用 **clusterExport()** 来将 **m** 发送到工作节点，之后作为一个全局变量供 **mtl()** 使用。

另外，调用 **options()** 是为了让 R 在我们做 “循环” 时不发出警告。一般我们并不这么做，但这里为了使用 **split()** 的需要。

之后，为了得到输出的列表中各个元素的总和，我们可以再次使用 **Reduce()**，但由于 R 的多样性，我们也可以使用 **do.call()** 函数。这个函数的动能正如其函数名暗示的：它会把列表 **counts** 中的每个元素抽出，之后作为参数传递给 **sum()**！（一般来说，当我们需要调用一个特定的函数，但其参数的数目直到运行时才可以确定时，**do.call()** 是非常有用的。）

正如前面所说的，除了使用 **split()**，我们可以将行数随机打乱：

```
1 tmp <- clusterSplit(cls,order(runif(nrow(m))))
```

这会为每一行产生一个 (0,1) 之间的随机数，之后按此排序。比如说，如果第三个随机数是第 20 小的，第三个元素在 **order()** 的输出中会是 20。这可以找到矩阵 **m** 行号的一个随机排列。

10.5.7 示例：邻接矩阵变换

这是4.13节中代码的 **snow** 版本。回忆一下，问题如下：

假如我们有一个图的邻接矩阵

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{10.1}$$

其中行号和列号从 0 开始，而不是 1。我们想要将其转换为一个两列的矩阵用来展示连接数，如下

所示

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 3 \\ 2 & 1 \\ 2 & 3 \\ 3 & 0 \\ 3 & 1 \\ 3 & 2 \end{pmatrix} \tag{10.2}$$

比如说，在上面的邻接矩阵中，最右边的第二行有一个 1，这意味着在顶点 1 和顶点 3 直接存在一条边。这个在转换后的矩阵中以 (1,3) 表示。

下面是在 **snow** 中进行该计算的代码：

```
tg <- function(cls,m) {
  n <- nrow(m)
  rowschunks <- clusterSplit(cls,1:n) # make chunks of row numbers
  m1 <- cbind(1:n,m) # prepend col of row numbers to m
  # now make the chunks of rows themselves
  tmp <- lapply(rowschunks,function(rchunk) m1[rchunk,])
  # launch the computation
  tmp <- clusterApply(cls,tmp,tgonchunk)
  do.call(rbind,tmp) # combine into one large matrix
}

# a worker works on a chunk of rows
tgonchunk <- function(rows) {
  # note: matrix space allocation not efficient
  mat <- NULL
  nc <- ncol(rows)
  for (i in 1:nrow(rows)) {
     row <- rows[i,]
     rownum <- row[1]
     for (j in 2:nc) {
        if (row[j] == 1) {
           if (is.null(mat)) {
              mat <- matrix(c(rownum,j-1),ncol=2)
           } else
              mat <- rbind(mat,c(rownum,j-1))
        }
     }
  }
  return(mat)
}
```

这里有什么新东西么？首先，由于我们需要最后的输出按字典序排列，我们需要保存原有每行的索引。所以我们需要在 **m** 中多添加一列：

```
m1 <- cbind(1:n,m) # prepend col of row numbers to m
```

其次，注意 **lapply()** 函数的使用。正如 **apply()** 会把一个特定的函数作用到矩阵的每一行（或每一列）上，**lapply()** 会把一个特定函数作用到列表的每一个元素上。输出结果依然是个列表。

在我们这里的例子中，我们需要将 **m** 按行分块传递给 **clusterApply()**，但后者要求我们必须传递一个列表。我们可以通过一个 **for** 来完成，一个一个地将分块添加进列表，但使用 **lapply()** 可以更加紧凑。

在最后，管理节点会接收新矩阵的很多部分，这些必须被整合起来。使用 **rbind()** 函数是很自然的想法，但我们仍然需要客服各个部分是 R 列表的问题。尽管 **Reduce()** 也可以完成，但用 **do.call()** 会更趁手。

需要注意的是，尽管在上一段中说使用 **rbind()** 是很自然的，但效率很低。这是因为 **rbind()** 会重新分配一个新的矩阵空间，这是个很浪费时间的操作。先分配 50 行空间，之后在构建矩阵时进行填充会是更好的选择。无论什么时候，我们用完了一个矩阵，我们都可以构建一个新的矩阵，之后把所有矩阵作为一个列表返回。

10.5.8 示例：设置节点 ID 和集群规模提示

让我们回忆一下，在 OpenMP 中有两个函数，**omp_get_thread_num()** 和 **omp_get_num_threads()**，分别用来报告一个线程的 ID 和线程总数。在 MPI 中，对应的函数时 **MPI_Comm_rank()** 和 **MPI_Comm_size()**。在 **snow** 中如果能有这样的函数（或功能），将是非常好的事情。这里的代码就是用于这个目的：

```
# sets a list myinfo as a global variable in the worker nodes in the
# cluster cls, with myinfo$id being the ID number of the worker and
# myinfo$nwrkrs being the number of workers in the cluster; called from
# the manager node
setmyinfo <- function(cls) {
   setmyinfo <- function(i,n) {
      myinfo <<- list(id = i, nwrkrs = n)
   }
   ncls <- length(cls)
   clusterApply(cls,1:ncls,setmyinfo,ncls)
}
```

是的，R 允许在函数中定义函数。顺便请注意超级赋值符<<-的使用，这个用了在全局层面进行赋值操作。

调用这个函数后，任何在一个工作节点上运行的代码代码都可以决定其节点 ID，比如在下面这样的代码中

```
if (myinfo$id == 1) ...
```

或者，我们也可以从管理节点传输代码到工作节点执行：

```
> setmyinfo(cls)
[[1]]
[[1]]$id
[1] 1

[[1]]$nwrkrs
[1] 2


[[2]]
[[2]]$id
[1] 2

[[2]]$nwrkrs
[1] 2

> clusterEvalQ(cls,myinfo$id)
[[1]]
[1] 1

[[2]]
[1] 2
```

第一个例子，由于 **clusterApply()** 有返回值，都会被打印出来。第二个例子中，调用

```
clusterEvalQ(cls,myinfo$id)
```

会使每个工作节点对表达式 **myinfo$id**进行求值；之后 **clusterEvalQ()** 返回在每个节点上的执行结果。

10.5.9 关闭集群

退出 R 之前，不要忘记使用 **stopCluster(clustername)** 来关闭集群。

10.6 multicore 扩展包

正如名字所暗示的，**multicore** 扩展包就是用于使用多核设备的计算能力的。这可能有点奇怪：既然 **snow** 既可以用于一个（物理）集群，也可以用于一个多核设备，而 **multicore** 只能在后者上使用，那用 **multicore** 的优势哪儿？答案是性能上的提高，这个在后面会解释。

这个扩展包的主函数是 **mclapply()**，其语法和 **snow** 中的 **clusterApply()** 很类似，也很类似地把任务分配给各个工作节点。

这里所说的工作节点，指的是同一台机器上的不同处理器。比如说，在一个四核的机器上运行 **multicore**，调用 **mclapply()** 会（默认）在你的机器上调用 4 个 R，并行地进行你的运算工作。其中每个 R 调用都使用和调用前的 R 一样的变量设置。因此所有的变量最初（注意这个修饰语）

是共享的，而不需要程序员采取特别的措施来将变量从管理节点分配到工作节点，这和 **snow** 相当不同。

这一切都是由 **mclapply()** 调用你操作系统中的 **fork()** 函数来完成的。（所以这仅限于类 Unix 系统，比如 Linux 和 MacOS。）这个 fork 过程是由 R 自己完成的，每个工作节点一个新的拷贝。

因此分配到 R 拷贝的工作节点会共享所有在 fork 发生时存在的变量（也包括你调用 **mclapply()** 时的局部变量）。所以你的代码不需要将这些变量拷贝到工作节点，工作节点会自动获取它们。但需要注意的是，这些变量只是在最初的时候是共享的，对其中一个拷贝的修改不会再其它拷贝中有所反应（包括最初的那个）。

从管理节点到工作节点的初始值拷贝是基于 **copy-on-write** 的，这意味着直到一个节点尝试获取数据，这份数据才会被拷贝过去。这个粒度（granularity）是在虚拟内存页（virtual memory page）层面（**??**节）上的。同样，这由操作系统处理，不是 R。

由于这个物理拷贝最终还是会由操作系统完成，所以 **multicore** 相对 **snow** 并没有很多人所想的那么有优势。然而，这可能在处理延迟方面由一定优势（**??**节）。有些情况下，不需要所有节点同时获取变量，所以可能一个节点在拷贝变量，而其余的在进行计算。.

还需要注意一点，**snow** 中，一个集群被设置好，会在每个 **snow** 函数调用中重复使用，而 **multicore** 与此不同，每一个 R 进程都在一个 **multicore** 函数被调用时从头开始。

10.6.1 示例：使用 multicore 进行邻接矩阵转换

和10.5.7节中的示例一样，而且实际上下面的 **tgonchunk()** 函数就是我们前面 **snow** 代码的修改版。

```
mclapply(starts,tgonchunk,m1,chunksize,mc.cores=ncores)
```

这个调用会将 **tgonchunk()** 函数作用于 **starts** 向量的每一个元素上（首先会被转换成 R 列表），其中 **m1** 和 **chunksize** 作为 **mclapply()** 的参数使用。

```
# transgraph problem, R multicore version

# arguments:
# m: the input matrix
# ncores: desired number of cores to use
tgmc <- function(m,ncores) {
  n <- nrow(m)
  chunksize <- floor(n/ncores)
  starts <- seq(1,n,chunksize)
  m1 <- cbind(1:n,m) # prepend col of row numbers to m
  tmp <- mclapply(starts,tgonchunk,m1,chunksize,mc.cores=ncores)
  do.call(rbind,tmp)
}

# a worker works on a chunk of rows
tgonchunk <- function(start,m1,chunksize) {
  # note: matrix space allocation not efficient
```

```
   outmat <- NULL
   end <- start + chunksize - 1
   nrm <- nrow(m1)
   if (end > nrm) end <- nrm
   ncm <- ncol(m1)
   for (i in start:end) {
      rownum <- m1[i,1]
      for (j in 2:ncm) {
         if (m1[i,j] == 1) {
            if (is.null(outmat)) {
               outmat <- matrix(c(rownum,j-1),ncol=2)
            } else
               outmat <- rbind(outmat,c(rownum,j-1))
         }
      }
   }
   return(outmat)
}
```

10.7 Rdsm

无论你在一个 NOW 网络还是一个多核机器上，我的 **Rdsm** 扩展包都可以作为一个多线程来使用。这是我在 2002 年开发的一个类似的 Perl 扩展包，PerlDSM[⑦]的扩展。**Rdsm** 的主要优势在于：

- 使用了一个内存共享的编程模型，正如在**??**节中所述，在并行编程社区中，一般认为这优于信息传递模型。
- 可以充分使用 R 的调试工具。

Rdsm 给了 R 程序员一个内存共享的视角，但实际上这些对象并没有共享。对象被储存在一个服务器上，通过网络端口获取[⑧]，从而使 R 程序员即使在 NOW 网络上也可以有一个类似多线程的视角。这里没有管理/工作节点的结构，所有的 R 进场都执行相同的代码。

Rdsm 中的共享对象，可以是 **dsmv** 和 **dsmm** 类中的数值向量或矩阵，也可以是 **dsml** 类中的 R 列表。为了效率，向量和矩阵与服务器的通讯是二进制的形式进行的，而列表进行了序列化。还有一个内置变量 **myinfo** 用于获取每一个进程的 ID 和进程总数，这和 **Rmpi** 中的 **mpi.comm.rank()** 和 **mpi.comm.size()** 返回的信息一样。

Rdsm 同样可以使用上面提到的 **install.packages()** 进行安装。**Rdsm** 提供了内置文档，不过最好还是要通读 **examples** 文件夹下的 **MatMul.R** 代码。里面提供了大量注释，希望可以作为这个扩展包的一个入门。

⑦N. Matloff, PerlDSM: A Distributed Shared Memory System for Perl, *Proceedings of PDPTA 2002*, 2002, 63-68.

⑧**Rdsm** 也可以在 **bigmemory** 扩展包中使用，见10.7.3节。

10.7.1 示例：使用 Rdsm 进行对角分块矩阵求逆

现在让我们来看如何将10.5.5节中的对角分块矩阵求逆使用 **Rdsm** 处理。

```
# invert a block diagonal matrix m, whose sizes are given in szs; here m
# is either an Rdsm or bigmemory shared variable; no return
# value--inversion is done in-place; it is assumed that there is one
# thread for each block

bdiaginv <- function(bd,szs) {
   # get number of rows of bd
   nrdb <- if(class(bd) == "big.matrix") dim(bd)[1] else bd$size[1]
   rownums <- getrng(nrdb,szs)
   myid <- myinfo$myid
   rng <- rownums[myid,1]:rownums[myid,2]
   bd[rng,rng] <- solve(bd[rng,rng])
   barr() # barrier
}

# find row number ranges for the blocks, returned in a 2-column matrix;
# matsz = number of rows in matrix, blkszs = block sizes
getrng <- function(matsz, blkszs) {
   nb <- length(blkszs)
   rwnms <- matrix(nrow=nb,ncol=2)
   for (i in 1:nb) {
      # i-th block will be in rows (and cols) i1:i2
      i1 <- if (i==1) 1 else i2 + 1
      i2 <- if (i == nb) matsz else i1 + blkszs[i] - 1
      rwnms[i,] <- c(i1,i2)
   }
   rwnms
}
```

相较于 **snow** 中的 11 行代码，这里主要的并行工作由这 4 行完成：

```
myid <- myinfo$myid
rng <- rownums[myid,1]:rownums[myid,2]
bd[rng,rng] <- solve(bd[rng,rng])
barr() # barrier
```

这也展示了内存共享编程模型相对信息传递模型的优势。

10.7.2 示例：Web Probe

在一般的编程社区中，一类主要的应用，甚至是在一个串行平台上，就是并行化 I/O。由于每一个 I/O 操作可以消耗很长时间（以 CPU 标准），如果可能的话，进行并行化是十分必要的。**Rdsm** 在 R 中提供了这样的功能。

下面的示例在一个很大的网站列表中循环进行，测量每次获取一个网站所用的时间。数据被储存在一个共享变量 **accesstimes** 中；前 **n** 个最近的获取时间被记录下来。每个 **Rdsm** 进程每次在处理一个网站。

这里的一个不寻常的特点是其中一个进程会马上退出，回到 R 的交换性命令行中。这就允许用户来检测搜集的数据。记住，共享的变量仍然可以被该进程获取。因此，当其他进程继续向 **accesstimes** 添加数据（每次添加时也进行一次删除），用户可以向退出的进程下命令来随着搜集工作，分析数据，比如说柱状图。

注意这里 lock/unlock 操作的使用，**Rdsm** 中使用了同样的名称。

```
# if the variable accesstimes is length n, then the Rdsm vector
# accesstimes stores the n most recent probed access times, with element
# i being the i-th oldest

# arguments:
# sitefile: IPs, one Web site per line
# ww: window width, desired length of accesstimes
webprobe <- function(sitefile,ww) {
  # create shared variables
  cnewdsm("accesstimes","dsmv","double",rep(0,ww))
  cnewdsm("naccesstimes","dsmv","double",0)
  barr() # Rdsm barrier
  # last thread is intended simply to provide access to humans, who
  # can do analyses on the data, typing commands, so have it exit this
  # function and return to the R command prompt
  # built-in R list myinfo has components to give thread ID number and
  # overall number of threads
  if (myinfo$myid == myinfo$nclnt) {
    print("back to R now")
    return()
  } else { # the other processes continually probe the Web:
    sites <- scan(sitefile,what="") # read from URL file
    nsites <- length(sites)
    repeat {
      # choose random site to probe
      site <- sites[sample(1:nsites,1)]
      # now probe it, recording the access time
      acc <- system.time(system(paste("wget --spider -q",site)))[3]
      # add to accesstimes, in sliding-window fashion
```

```
30          lock("acclock")
31          if (naccesstimes[1] < ww) {
32             naccesstimes[1] <- naccesstimes[1] + 1
33             accesstimes[naccesstimes[1]] <- acc
34          } else {
35             # out with the oldest, in with the newest
36             newvec <- c(accesstimes[-1],acc)
37             accesstimes[] <- newvec
38          }
39          unlock("acclock")
40       }
41    }
42 }
```

10.7.3 bigmemory 扩展包

Jay Emerson 和 Mike Kane 在我开发 **Rdsm** 的同时，开发了 **bigmemory** 扩展包；而我们互相都不知道这一点。

bigmemory 扩展包的目的不在于提供一个多线程环境。它的目的在于处理一个 R 的硬性限制：任何 R 对象都不能大于 $2^{31}-1$ 字节。即使你用一个有很大内存的 64 位机器，这个限制也是存在的。**bigmemory** 扩展包通过使用操作系统的调用在进程间设置共享内存，从而在多核机器上解决了这个问题[⑨]。

理论上讲，**bigmemory** 也可以用在多线程上，但其为包含这方面的机制。然而，**Rdsm** 可以和 **bigmemory** 一起使用，由于后者的高效，这也带来了优势。

在 **Rdsm** 中使用 **bigmemory** 变量非常简单：使用 **newbm()** 而不是 **cnewdsm()** 来创建共享变量即可。

10.8 R 和 GPU

未来几年中，（为了特定问题）将 GPU 的高效和 R 进行结合一定会让越来越多的程序员感兴趣。

现在，进行 GPU 开发的主流框架就是在 NVIDIA 的显卡上使用 CUDA。CUDA 是 C 的一个扩展。

如果你需要写自己的 CUDA 代码，那你可能要使用10.9节中的方法。但在多数情况下，你可以从 R 中 GPU 计算的两个主要扩展包，**gputools** 和 **rgpu**，找到你需要的功能。两个扩展包都主要处理线性代数操作。这小节剩余部分会讨论这两个扩展包。

10.8.1 安装

由于两个扩展包中链接到 CUDA 库的问题，你可能不能通过 **install.packages()** 来进行安装。我推荐的安装方法如下：

- 下载**.tar.gz** 格式的扩展包。

[⑨]通过使用操作系统将内存映射到文件上，这个扩展包也可以在分布式系统上使用

- 将扩展包解压缩，我们把产生的文件夹叫做 **x**。
- 假设你想把它安装到**/a/b/c**。
- 对 **x** 中的文件进行修改。
- 之后运行 `R CMD INSTALL -l /a/b/c x`。

更多细节会在后面的章节中讨论。

10.8.2 gputools 扩展包

为了安装 **gputools**，我从 CRAN 下载了源代码，并像前面提到的解压缩。我去掉了 **src** 文件夹下 **Makefile.in** 文件中的几个选项 `-gencode arch=compute_20,code=sm_20`。我还确定了 shell 的启动文件中包含了 CUDA 的执行路径和库路径，**/usr/local/cuda/bin** 和**/usr/local/cuda/lib**。

之后我运行了 **R CMD INSTALL**。我使用了 **gpuLm.fit()** 进行测试，R 中 **lm.fit()** 在 **gputools** 中的对应版本。

这个扩展包提供了多种线性代数操作，比如矩阵相乘、求解 $Ax = b$（矩阵求逆）和奇异值分解，以及一些需要大量计算的操作，比如线性/广义线性模型估计和层级聚类。

这里是如何求矩阵 **m** 平方的示例：

```
> m2 <- gpuMatMult(m,m)
```

gpuSolve() 函数和 R 中的 **solve()** 一样。调用 `gpuSolve(a,b)`会求解线性系统 $ax = b$，其中 **a** 是一个方阵，而 **b** 是一个向量。如果第二个参数缺失，会返回 a^{-1}。

10.8.3 rgpu 扩展包

为了安装 **rgpu**，我从 `https://gforge.nbic.nl/frs/?group_id=38`下载源代码并解压缩。之后我修改了 **Makefile** 文件中的几行 [10]

```
LIBS = -L/usr/lib/nvidia -lcuda -lcudart -lcublas
CUDA_INC_PATH = /home/matloff/NVIDIA_GPU_Computing_SDK/C/common/inc
R_INC_PATH = /usr/include/R
```

第一行是为了让系统找到 **-lcuda**，这点和 **gputools** 一样。第二行是为了 NVIDIA SDK 中的 **cutil.h** 文件，上面的是我的安装路径。

第三行中，我生成了一个 **z.c** 文件，其中只包含

```
#include <R.h>
```

一行，之后运行

```
R CMD SHLIB z.c
```

来看 R 的引用文件究竟在哪里。

在 2010 年 5 月是，**rgpu** 中的函数远少于 **gputools**。然而，**rgpu** 中一个很好的特性在于在进行矩阵运算时，不需要将中间结果从 device 内存返回到 host 内存，这是个开销很大的操作。这里展示了如何计算矩阵 **m** 的乘法，并加上自身：

[10] 译者注：请根据自己机器上的相应路径进行修改

```
> m2m <- evalgpu(m %*% m + m)
```

10.9 通过在 R 中调用 C 进行并行

并行 R 的目的在于比普通的 R 要快。但即使这个目的达到了，这也还是 R，也就是说，还是可能很慢。

人们总是必须决定花费多少精力在优化上面的。为了最快的速度，我们甚至都不应该用 C，而应该用汇编语言。类似地，也必须决定是纯粹用 R，还是用更快的 C。如果并行 R 给了你所需要的速度，那再好不过；如果速度不够，你应该考虑在主体仍用 R 的情况下，用 C 完成一部分工作。你会发现，在保持用 R 的方便的前提下，在用 C 处理代码的并行部分已经足够好了。

10.9.1 在 R 中调用 C

在 C 中，二维数组以行序存储，和 R 中列序相反。例如，如果我们有一个 3×4 的数组，第二行第二列中的元素在线性视角下是第 5 个元素，因为第一列中有 3 个元素，而这是第二列中的第二个元素。当然，还需要注意 C 的计数从 0 开始，而 R 从 1 开始。在写供 R 使用的 C 代码时，你必须考虑这些问题。

所有从 R 传递到 C 的参数都被作为 C 的指针。注意 C 函数自身必须返回 `void`。在 R/C 中传递的数值比如作为函数的参数，比如我们下面示例中的 `result`。

10.9.2 示例：矩阵的次对角线

作为一个示例，这里是用于求一个方阵次对角线的 C 代码[1]。代码被保存在一个名为 **sd.c** 的文件中：

```
// arguments:
// m: a square matrix
// n: number of rows/columns of m
// k: the subdiagonal index--0 for main diagonal, 1 for first
// subdiagonal, 2 for the second, etc.
// result: space for the requested subdiagonal, returned here

void subdiag(double *m, int *n, int *k, double *result)
{
  int nval = *n, kval = *k;
  int stride = nval + 1;
  for (int i = 0, j = kval; i < nval-kval; ++i, j+= stride)
     result[i] = m[j];
}
```

为了方便，你可以在一个命令行中运行 R 来编译它，这会调用 GCC：

```
% R CMD SHLIB sd.c
gcc -std=gnu99 -I/usr/share/R/include -fpic -g -O2 -c sd.c -o sd.o
```

[1] 感谢我的研究生助理 Min-Yu Huang，他完成了这个函数的早期版本。

```
gcc -std=gnu99 -shared -o sd.so sd.o -L/usr/lib/R/lib -lR
```

注意 R 向我们展示了它在调用 GCC 时所作的具体操作。这允许我们来做一些改动。

需要注意这只会生成一个动态链接库 **sd.o**，而不是一个可执行程序。（在 Windows 下会是**.dll**）所以，怎么来执行它？答案是使用 R 的 **dyn.load()** 函数来载入它。这里是一个示例：

```
> dyn.load("sd.so")
> m <- rbind(1:5, 6:10, 11:15, 16:20, 21:25)
> k <- 2
> .C("subdiag", as.double(m), as.integer(dim(m)[1]), as.integer(k),
result=double(dim(m)[1]-k))
[[1]]
 [1] 1 6 11 16 21 2 7 12 17 22 3 8 13 18 23 4 9 14 19 24 5 10 15 20 25

[[2]]
[1] 5

[[3]]
[1] 2

$result
[1] 11 17 23
```

注意我们需要在调用时为 **result** 分配内存空间。从上面的结果来看，我们的函数在对应空间中放置的值是正确的。T

10.9.3 在 R 中调用 OpenMPI C 代码

由于 OpenMP 可以由 C 使用，这就使得其可以从 R 中调用。（关于 OpenMP 的详细讨论请见第4章。）

在10.9节中类似，代码被编译并载入到 R 会话，尽管有一些额外的步骤用于在调用 GCC 时设置 `-fopenmp` 参数（你需要手动运行，而不是使用 **R CMD SHLIB**）。

10.9.4 在 R 中调用 CUDA 代码

这里也适用同样的原则，但需要小心调用的库和类似的问题。

和上面一样，我们需要编译以动态链接库而不是可执行文件。这是下面要用的文件 **mutlinksforr.cu**，和编译用的命令：

```
pc41:~% nvcc -g -G -I/usr/local/cuda/include -Xcompiler
   "-I/usr/include/R -fpic" -c mutlinksforr.cu -o mutlinks.o -arch=sm_11
pc41:~% nvcc -shared -Xlinker "-L/usr/lib/R/lib -lR"
   -L/usr/local/cuda/lib mutlinks.o -o meanlinks.so
```

这会生成 **meanlinks.so**。之后我在 R 中进行测试：

```
> dyn.load("meanlinks.so")
```

```
> m <- rbind(c(0,1,1,1),c(1,0,0,1),c(1,0,0,1),c(1,1,1,0))
> ma <- rbind(c(0,1,0),c(1,0,0),c(1,0,0))
> .C("meanout",as.integer(m),as.integer(4),mo=double(1))
[[1]]
 [1] 0 1 1 1 1 0 0 1 1 0 0 1 1 1 1 0

[[2]]
[1] 4

$mo
[1] 1.333333

> .C("meanout",as.integer(ma),as.integer(3),mo=double(1))
[[1]]
[1] 0 1 1 1 0 0 0 0 0

[[2]]
[1] 3

$mo
[1] 0.3333333
```

10.9.5 示例：Mutual Outlink

我们再次使用 2.4.3节中的 Mutual Outlink 示例。这里是 R/CUDA 版本的代码：

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

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

// the following is needed to avoid variable name mangling
extern "C" void meanout(int *hm, int *nrc, double *meanmut);

// for a given thread number tn, calculates 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;
   // make sure to account for R being column-major order; R's i-th row
   // is our i-th column here
   int startrowa = pair[0],
       startrowb = pair[1];
   for (int k = 0; k < n; k++)
      sum += m[startrowa + n*k] * m[startrowb + n*k];
   atomicAdd(tot,sum);
}

// meanout() is called from R
// hm points to the link matrix, nrc to the matrix size, meanmut to the output
void meanout(int *hm, int *nrc, double *meanmut)
{
    int n = *nrc,msize=n*n*sizeof(int);
    int *dm, // device matrix
        htot, // host grand total
        *dtot; // device grand total
    cudaMalloc((void **)&dm,msize);
    cudaMemcpy(dm,hm,msize,cudaMemcpyHostToDevice);
    htot = 0;
    cudaMalloc((void **)&dtot,sizeof(int));
    cudaMemcpy(dtot,&htot,sizeof(int),cudaMemcpyHostToDevice);
    dim3 dimGrid(1,1);
    int npairs = n*(n-1)/2;
    dim3 dimBlock(npairs,1,1);
    proc1pair<<<dimGrid,dimBlock>>>(dm,dtot,n);
    cudaThreadSynchronize();
```

```
61     cudaMemcpy(&htot,dtot,sizeof(int),cudaMemcpyDeviceToHost);
62     *meanmut = htot/double(npairs);
63     cudaFree(dm);
64     cudaFree(dtot);
65 }
```

这份代码几乎没有进行优化。比如，我们应该在每一个 block 中使用不止一个线程。

10.10 调试 R 程序

R 内置的调试机制是首选，在还存在着其它选择。

10.10.1 文本编辑器

然而，如果你是一个 Vim 编辑器的粉丝，我开发了一个可以极大扩展 R 调试器的工具。请从 R 的 CRAN 上下载 **edtdbg**。Emacs 中也有类似的工具。

Vitalie Spinu 的 `ess-tracebug` 运行于 Emacs。它大体基于 `edtdbg`，但提供了更多的针对 Emacs 的特性。

10.10.2 IDE

我个人不是提倡使用 IDE，但的确有一些很优秀的 IDE。

REvolution Analytics，一家提供 R 咨询和再开发版本 R 的公司，他们提供了一个包含了很好的调试机制的 IDE。但它只可以在 Windows 上运行，而且必须安装 Microsoft Visual Studio。

StatET，一个基于 Eclipse 的跨平台 IDE 的开发者在 2011 年五月添加了调试工具。

RStudio，另一个跨平台的 IDE 的开发者，从 2011 年夏天也开始计划添加调试器[12]。

10.10.3 缺少命令行终端的问题

诸如 **Rmpi**、**snow**、**foreach** 和其它的并行 R 扩展包并未给每一个进程设置命令行终端，从而使得在工作节点上进行调试变得不可能。那我们怎样调试使用这些扩展包的程序呢？这里拿 **snow** 做个例子。

首先，需要调试调试每个工作节点上的函数，比如 10.5.6节 mutual outlink 示例中的 **mtl()** 函数。这时需要人为设置一下参数的值，之后使用 R 常规调试机制。

这可能有效。但 bug 很可能就出现在参数本身上，或者出现在我们设置它们的方式上。事情就变得困难了。由于 **print()** 在工作进程中无法工作，即使打印出诸如变量值的追踪信息都很难。**message()** 函数可能对一些扩展包有效；但如果无效，你需要自己使用 **cat()** 来将变量写到文件中。

Rdsm 支持全面的调试，它在每个进程中都有一个单独的命令行终端。

10.10.4 调试 R 所调用的 C 代码

如果像10.9节中，并行是通过在 R 中调用 C，生成一个动态链接库实现的，调试会更复杂一些。首先，需要在 GDB 下启动 R，之后载入需要调试的库。这是 R 的解释器会循环读取你发出的

[12]译者注：RStudio 中的调试功能已添加

R 命令。通过使用 ctrl-c 来跳出循环，这会让你返回 **GDB** 的解释器。之后在需要调试的 C 函数，比如上面例子中的 **subdiag()**，设置断点。最终，告诉 GDB 继续，它就在你的函数中暂停下来。这里展示了你的会话内容：

```
$ R -d gdb
GNU gdb 6.8-debian
...
(gdb) run
Starting program: /usr/lib/R/bin/exec/R
...
> dyn.load("sd.so")
```

10.11 本书中的其它 R 语言示例

见下列章节中的示例（一些是非并行的）：
12.5.4节、14.2.1节（非并行）和14.5.1节（非并行）。

- 线性等式的并行 Jacobi 迭代，12.5.4节。
- 1 维 FFT 的矩阵运算，14.2.1节（可以通过并行的矩阵相乘来并行化）。
- 2 维 FFT 的并行计算，14.4.1节。
- 图像平滑，14.5.1节。

第 11 章　The Parallel Prefix Problem

An operation that arises in a variety of parallel algorithms is that of *prefix* (or *scan*). In its abstract form, it inputs a sequence of objects $(x_0, ..., x_{n-1})$, and outputs $(s_0, ..., s_{n-1})$, where

$$\begin{aligned} s_0 &= x_0, \\ s_1 &= x_0 \otimes x_1, \\ &..., \\ s_{n-1} &= x_0 \otimes x_1 \otimes ... \otimes x_{n-1} \end{aligned} \tag{11.1}$$

where $\otimes$ is some associative operator.

That's pretty abstract. The most concrete example would be that in which $\otimes$ is + and the objects are numbers. The scan of (12,5,13) would then be (12,12+5,12+5+13) = (12,17,30).

This is called an **inclusive** scan, in which x_i is included in s_i. The **exclusive** version of the above example would be (0,12,17).

Prefix scan has become a popular tool in the parallel processing community, applicable in a surprising variety of situations. Various examples will arise in succeeding chapters, but we'll present one in the next section in order to illustrate the versatility of the prefix approach.

11.1　Example: Permutations

Say we have the vector (12,5,13,8,88). Applying the permutation (2,0) would say the old element 0 becomes element 2, the old element 2 becomes element 0, and all the rest stay the same. The result would be (13,5,12,8,88). If we then applied the permutation (1,2,4), it would mean that element 1 goes to position 2, 2 goes to 4, and 4 goes to 1, with everything else staying put. Our new vector would then be (13,88,5,8,12).

This too can be cast in matrix terms, by representing any permutation as a matrix multiplication. We just apply the permutation to the identity matrix I, and then postmultiply the (row) vector by the matrix. For instance, the matrix corresponding to the permutation (1,2,4) is

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{11.2}$$

so applying (1,2,4) to (12,5,13,8,88) above can be done as

$$(12,5,13,8,88)\begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} = (13,5,12,8,88) \tag{11.3}$$

So in terms of (11.1), x_0 would be the identity matrix, x_i for i > 0 would be the ith permutation matrix, and $\otimes$ would be matrix multiplication.

Note, however, that although we've couched the problem in terms of matrix multiplication, these are *sparse* matrices, i.e. have many 0s. Thus a general parallel matrix-multiply routine may not be efficient, and special parallel methods for sparse matrices should be used (Section 12.7).

Note that the above example shows that in finding a scan,

- the elements might be nonscalars
- the associative operator need not be commutative

11.2 General Strategies for Parallel Scan Computation

For the time being, we'll assume we have n threads, i.e. one for each datum. Clearly this condition will often not hold, so we'll extend things later.

We'll describe what is known as a *data parallel* solution to the prefix problem.

Here's the basic idea, say for n = 8:

Step 1:

$$x_1 \leftarrow x_0 + x_1 \tag{11.4}$$
$$x_2 \leftarrow x_1 + x_2 \tag{11.5}$$
$$x_3 \leftarrow x_2 + x_3 \tag{11.6}$$
$$x_4 \leftarrow x_3 + x_4 \tag{11.7}$$
$$x_5 \leftarrow x_4 + x_5 \tag{11.8}$$
$$x_6 \leftarrow x_5 + x_6 \tag{11.9}$$
$$x_7 \leftarrow x_6 + x_7 \tag{11.10}$$

Step 2:

$$x_2 \leftarrow x_0 + x_2 \tag{11.11}$$
$$x_3 \leftarrow x_1 + x_3 \tag{11.12}$$
$$x_4 \leftarrow x_2 + x_4 \tag{11.13}$$
$$x_5 \leftarrow x_3 + x_5 \tag{11.14}$$
$$x_6 \leftarrow x_4 + x_6 \tag{11.15}$$
$$x_7 \leftarrow x_5 + x_7 \tag{11.16}$$

Step 3:

$$x_4 \leftarrow x_0 + x_4 \tag{11.17}$$

$$x_5 \leftarrow x_1 + x_5 \tag{11.18}$$

$$x_6 \leftarrow x_2 + x_6 \tag{11.19}$$

$$x_7 \leftarrow x_3 + x_7 \tag{11.20}$$

In Step 1, we look at elements that are 1 apart, then Step 2 considers the ones that are 2 apart, then 4 for Step 3.

Why does this work? Well, consider how the contents of x_7 evolve over time. Let a_i be the original x_i, i = 0,1,...,n-1. Then here is x_7 after the various steps:

step	contents
1	$a_6 + a_7$
2	$a_4 + a_5 + a_6 + a_7$
3	$a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7$

Similarly, after Step 3, the contents of x_7 will be $a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6$ (check it!). So, in the end, the locations of x_i will indeed contain the prefix sums.

For general n, the routing is as follows. At Step i, each x_j is routed both to itself and to $x_{j+2^{i-1}}$, for $j >= 2^{i-1}$. (Some threads, more in each successive step, are idle.)

There will be $log_2 n$ steps, or if n is not a power of 2, case the number of steps is $\lfloor log_2 n \rfloor$.

Note two important points:

- The location x_i appears both as an input and an output in the assignment operations above. In our implementation, we need to take care that the location is not written to before its value is read. One way to do this is to set up an auxiliary array y_i. In odd-numbered steps, the y_i are written to with the x_i as inputs, and vice versa for the even-numbered steps.
- As noted above, as time goes on, more and more threads are idle. Thus load balancing is poor.
- Synchronization at each step incurs overhead in a multicore/multiprocessr setting. (Worse for GPU if multiple blocks are used).

Now, what if n is greater than p, our number of threads? Let Ti denote thread i. The standard approach is that taken in Section 5.10:

```
1 break the array into p blocks
2 parallel for i = 0,...,p-1
3   Ti does scan of block i, resulting in Si
4 form new array G of rightmost elements of each Si
5 do parallel scan of G
6 parallel for i = 1,...,p-1
7   Ti adds Gi to each element of block i
```

For example, say we have the array

```
1 2 25 26 8 50 3 1 11 7 9 29 10
```

and three threads. We break the data into three sections,

2 25 26 8	50 3 1 11	7 9 29 10

and then apply a scan to each section:

2 27 53 61	50 53 54 65	7 16 45 55

But we still don't have the scan of the array overall. That 50, for instance, should be 61+50 = 111 and the 53 should be 61+53 = 114. In other words, 61 must be added to that second section, (50,53,54,65), and 61+65 = 126 must be added to the third section, (7,16,45,55). This then is the last step, yielding

2 27 53 61	111 114 115 126	133 142 171 181

Another possible approach would be make n "fake" threads FTj. Each Ti plays the role of n/p of the FTj. The FTj then do the parallel scan as at the beginning of this section. Key point: Whenever a Ti becomes idle, it is assigned to help other Tk.

11.3 Implementations

The MPI standard actually includes built-in parallel prefix functions, **MPI_Scan()**. A number of choices are offered for $\otimes$, such as maximum, minimum, sum, product etc.

The Thrust library for CUDA or OpenMP includes functions **thrust::inclusive_scan()** and **thrust::exclusive_scan()**.

The CUDPP (CUDA Data Parallel Primitives Library) package contains CUDA functions for sorting and other operations, many of which are based on parallel scan. See `http://gpgpu.org/developer/cudpp` for the library code, and a detailed analysis of optimizing parallel prefix in a GPU context in the book *GPU Gems 3*, available either in bookstores or free online at `http://developer.nvidia.com/object/gpu_gems_home.html`.

11.4 Example: Parallel Prefix, Run-Length Decoding in OpenMP

Here an OpenMP implementation of the approach described at the end of Section 11.2, for addition:

```
1  #include <omp.h>
2
3  // calculates prefix sums sequentially on u, in-place, where u is an
4  // m-element array
5  void seqprfsum(int *u,int m)
6  { int i,s=u[0];
7    for (i = 1; i < m; i++) {
8       u[i] += s;
9       s = u[i];
10   }
11 }
12
13 // OMP example, calculating prefix sums in parallel on the n-element
14 // array x, in-place; for simplicity, assume that n is divisible by the
```

```
// number of threads; z is for intermediate storage, an array with length
// equal to the number of threads; x and z point to global arrays
void parprfsum(int *x, int n, int *z)
{
   #pragma omp parallel
   { int i,j,me = omp_get_thread_num(),
         nth = omp_get_num_threads(),
         chunksize = n / nth,
         start = me * chunksize;
      seqprfsum(&x[start],chunksize);
      #pragma omp barrier
      #pragma omp single
      {
      for (i = 0; i < nth-1; i++)
         z[i] = x[(i+1)*chunksize - 1];
      seqprfsum(z,nth-1);
      }
      if (me > 0) {
         for (j = start; j < start + chunksize; j++) {
            x[j] += z[me - 1];
         }
      }
   }
}
```

Here is an example of use: A method for compressing data is to store only repeat counts in *runs*, where the latter means a set of consecutive, identical values. For instance, the sequence 2,2,2,0,0,5,0,0 would be compressed to 3,2,2,0,1,5,2,0, meaning that the data consist of first three 2s, then two 0s, then one 5, and finally two 0s. Note that the compressed version consists of alternating *run counts* and *run values*, respectively 2 and 0 at the end of the above example.

To solve this in OpenMP, we'll first call the above functions to decide where to place the runs in our overall output.

```
void uncomprle(int *x,int nx,int *tmp,int *y,int *ny)
{
   int i,nx2 = nx/2;
   int z[MAXTHREADS];
   for (i = 0; i < nx2; i++) tmp[i+1] = x[2*i];
   parprfsum(tmp+1,nx2+1,z);
   tmp[0] = 0;
   #pragma omp parallel
   { int j,k;
      int me=omp_get_thread_num();
      #pragma omp for
```

```
12      for (j = 0; j < nx2; j++) {
13         // where to start the j-th run?
14         int start = tmp[j];
15         // what value is in the run?
16         int val = x[2*j+1];
17         // how long is the run?
18         int nrun = x[2*j];
19         for (k = 0; k < nrun; k++)
20            y[start+k] = val;
21      }
22   }
23   *ny = tmp[nx2];
24 }
```

11.5 Example: Run-Length Decompression in Thrust

Here's how we could do the first part of the operation above, i.e. determining where to place the runs in our overall output, in Thrust:

```
1  #include <stdio.h>
2  #include <thrust/device_vector.h>
3  #include <thrust/scan.h>
4  #include <thrust/sequence.h>
5  #include <thrust/remove.h>
6
7  struct iseven {
8    bool operator()(const int i)
9    { return (i % 2) == 0;
10   }
11 };
12
13 int main()
14 { int i;
15   int x[12] = {2,3,1,9,3,5,2,6,2,88,1,12};
16   int nx = 12;
17   thrust::device_vector<int> out(nx);
18   thrust::device_vector<int> seq(nx);
19   thrust::sequence(seq.begin(),seq.end(),0);
20   thrust::device_vector<int> dx(x,x+nx);
21   thrust::device_vector<int>::iterator newend =
22      thrust::copy_if(dx.begin(),dx.end(),seq.begin(),out.begin(),iseven());
23   thrust::inclusive_scan(out.begin(),out.end(),out.begin());
24   // "out" should be 2,2+1 = 3,2+1+3=6,...
```

```
25    thrust::copy(out.begin(), newend,
26       std::ostream_iterator<int>(std::cout, " "));
27    std::cout << "\n";
28 }
```

第 12 章　Introduction to Parallel Matrix Operations

12.1　"We're Not in Physicsland Anymore, Toto"

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 toys 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, such as image processing, social networks and data mining. Google employs a number of linear algebra experts, and they deal with matrices with literally millions of rows or columns.

We assume for now that the matrices are **dense**, meaning that most of their entries are nonzero. This is in contrast to **sparse** matrices, with many zeros. Clearly we would use differents type of algorithms for sparse matrices than for dense ones. We'll cover sparse matrices a bit in Section 12.7.

12.2　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

$$A = \begin{pmatrix} 1 & 5 & 12 \\ 0 & 3 & 6 \\ 4 & 8 & 2 \end{pmatrix} \tag{12.1}$$

and

$$B = \begin{pmatrix} 0 & 2 & 5 \\ 0 & 9 & 10 \\ 1 & 1 & 2 \end{pmatrix}, \tag{12.2}$$

so that

$$C = AB = \begin{pmatrix} 12 & 59 & 79 \\ 6 & 33 & 42 \\ 2 & 82 & 104 \end{pmatrix}. \tag{12.3}$$

We could partition A as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}, \tag{12.4}$$

where

$$A_{00} = \begin{pmatrix} 1 & 5 \\ 0 & 3 \end{pmatrix}, \tag{12.5}$$

$$A_{01} = \begin{pmatrix} 12 \\ 6 \end{pmatrix}, \tag{12.6}$$

$$A_{10} = \begin{pmatrix} 4 & 8 \end{pmatrix} \tag{12.7}$$

and

$$A_{11} = \begin{pmatrix} 2 \end{pmatrix}. \tag{12.8}$$

Similarly we would partition B and C into blocks of a compatible size to A,

$$B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \tag{12.9}$$

and

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}, \tag{12.10}$$

so that for example

$$B_{10} = \begin{pmatrix} 1 & 1 \end{pmatrix}. \tag{12.11}$$

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

$$C_{00} = A_{00}B_{00} + A_{01}B_{10}, \tag{12.12}$$

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}.

12.3 Parallel Matrix Multiplication

Since so many parallel matrix algorithms rely on matrix multiplication, a core issue is how to parallelize that operation.

Let's suppose for the sake of simplicity that each of the matrices to be multiplied is of dimensions n x n. Let p denote the number of "processes," such as shared-memory threads or message-passing nodes.

12.3.1 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.

Fox's Algorithm

Consider the node that has the responsibility of calculating block (i,j) of the product C, which it calculates as

$$A_{i0}B_{0j} + A_{i1}B_{1j} + ... + A_{ii}B_{ij} + ... + A_{i,m-1}B_{m-1,j} \tag{12.13}$$

Rearrange this with A_{ii} first:

$$A_{ii}B_{ij} + A_{i,i+1}B_{,i+1j} + ... + A_{i,m-1}B_{m-1,j} + A_{i0}B_{0j} + A_{i1}B_{1j} + ... + A_{i,i-1}B_{i-1,j} \tag{12.14}$$

Written more compactly, this is

$$\sum_{k=0}^{m-1} A_{i,(i+k)mod\ m}B_{(i+k)mod\ m,j} \tag{12.15}$$

In other words, start with the A_{ii} term, then go across row i of A, wrapping back up to the left end when you reach the right end. The order of summation in this rearrangement will be the actual order of computation. It's similar for B, in column j.

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):

```
1 iup = i+1 mod m;
2 idown = i-1 mod m;
3 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]
}
```

The main idea is to have the various computational nodes repeatedly exchange submatrices with each other, timed so that a node receives the submatrix it needs for its computation "just in time."

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.

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.

12.3.2 Shared-Memory Case

Example: Matrix Multiply in 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.

Example: Matrix Multiply in 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, float *total)
{ int k,i,j; float sum;
   // find i,j according to thread and block ID
   sum = 0;
   for (k = 0; k < ncolsa; k++)
      sum += a[i*ncolsa+k] * b[k*ncols+j];
  *total = sum;
}
```

This should produce a good speedup. But we can do even better, much much better.

The CUBLAS package includes very finely-tuned algorithms for matrix multiplication. The CUBLAS source code is not public, though, so in order to get an idea of how such tuning might be done, let's look at Prof. Richard Edgar's algorithm, which makes use of shared memory. (Actually, this may be what CUBLAS uses.)

```
__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();
```

```
29      // Perform multiplication on sub-matrices
30      // Each thread computes one element of the C sub-matrix
31      for( int k = 0; k < BLOCK_SIZE; k++ ) {
32         Csub += As[ty][k] * Bs[k][tx];
33      }
34     // Synchronise again
35     __syncthreads();
36    }
37    // Write the C sub-matrix back to global memory
38    // Each thread writes one element
39    int c = (dc_wB * BLOCK_SIZE * by) + (BLOCK_SIZE*bx);
40    C[c + (dc_wB*ty) + tx] = Csub;
41 }
```

Here are the relevant portions of the calling code, including defined constants giving the number of columns ("width") of the multiplier matrix and the number of rows ("height") of the multiplicand:

```
1  #define BLOCK_SIZE 16
2  ...
3  __constant__ int dc_wA;
4  __constant__ int dc_wB;
5  ...
6  // Sizes must be multiples of BLOCK_SIZE
7  dim3 threads(BLOCK_SIZE,BLOCK_SIZE);
8  dim3 grid(wB/BLOCK_SIZE,hA/BLOCK_SIZE);
9  MultiplySimple<<<grid,threads>>>(d_A, d_B, d_C);
10 ...
```

(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

```
1  for(int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b+= bStep) {
2    // 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

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix} \tag{12.16}$$

and

$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \end{pmatrix} \tag{12.17}$$

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_{00}, the 2x2 submatrix of C's upper-left corner. Due to the fact that partitioned matrices multiply "just like numbers," we have

$$C_{00} = A_{00}B_{00} + A_{01}B_{10} + A_{02}B_{20} \tag{12.18}$$

$$= \begin{pmatrix} 1 & 2 \\ 7 & 8 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} + ... \tag{12.19}$$

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

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \end{pmatrix} = 11 \tag{12.20}$$

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!

12.3.3 R Snow

Section 1.3.4 showed how to parallelize a matrix-vector product computation in **snow**, by breaking the matrix rows into chunks, and then exploiting the tiling properties of matrices. Computation of matrix-matrix products can be done in the same way.

12.3.4 R Interfaces to GPUs

The most widely used of these is probability the **gputools** library. It includes various matrix routines, including **gpuMatMult()** for matrix multiplication.

12.4 Finding Powers of Matrices

In some applications, we are interested not just in multiplying two matrices, but rather in multiplying a matrix by itself, many times.

12.4.1 Example: Graph Connectedness

Let n denote the number of vertices in the graph. As before, 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.

Our ultimate goal here will be to compute the corresponding **reachability matrix** $R^{(k)}$ has its (i,j) element equal to 1 if there is some path from i to j taking k or fewer steps, and 0 otherwise. (Note that the notation "(k)" here is a superscript, not an exponent.) We would especially like to compute R, whose elements indicate whether one can *ever* reach one vertex starting at another. In particular, we may be interested in determining whether the graph is **connected**, meaning that every vertex eventually leads to every other vertex.

Toward that end, consider the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \tag{12.21}$$

Let's take our row/column numbering convention to start at 1, not 0.

Let's ask the question, Can we get from vertex 3 to vertex 1 in two steps? The answer is yes; indeed, there are two such paths:

$$3 \rightarrow 2 \rightarrow 1 \tag{12.22}$$

$$3 \rightarrow 4 \rightarrow 1 \tag{12.23}$$

If we were to answer this kind of question systematically, say for the number of two-step paths from i to j, we would evaluate the following boolean expression:

$$p(i \rightarrow 1 \rightarrow j) + p(i \rightarrow 2 \rightarrow j) + p(i \rightarrow 3 \rightarrow j) + p(i \rightarrow 4 \rightarrow j) \tag{12.24}$$

where $p()$ is equal to 1 if the postulated path exists, 0 if not.

But observe that

$$p(i \rightarrow k \rightarrow j) = a_{ik} \cdot a_{kj} \tag{12.25}$$

Thus the number of paths for a general $n \times n$ matrix A from vertex i to vertex j is

$$\sum_{i=1}^{n} a_{ik} \cdot a_{kj} \tag{12.26}$$

But this is the the (i,j) element of A^2! Moreover, this says that $R^{(2)} = b(A^2)$, where **b()** changes nonzero elements of a matrix to 1s, and retains the original 0s.

In general:

Theorem 1 *Suppose A is the adjacency matrix A for a graph. Then*

(a) The number of r-step paths from i to j is the (i,j) element of A^r.

(b)

$$R^{(k)} = b(A^k) \tag{12.27}$$

(c) Since the longest possible distinct path has length n-1, we have that the graph is connected if and only if each of the matrices $R^{(1)}, ..., R^{(n-1)}$ has all of its off-diagonal elements equal to 1.

(d) Suppose the graph is undirected. Then **cycles** *are possible, so we can keep coming back to a vertex. Thus the graph is connected if and only if some matrix among $R^{(1)}, ..., R^{(n-1)}$ has all of its off-diagonal elements equal to 1.*

So, the original graph connectivity problem reduces to a matrix problem. And (d) is especially interesting, as it means that if we do manage to find some $R^{(k)}$ that consists of all 1s (off the diagonal), our computation is done.

12.4.2 Example: Fibonacci Numbers

The basic problem is well known: Find the Fibonacci numbers f_n, where

$$f_0 = f_1 = 1 \tag{12.28}$$

and

$$f_n = f_{n-1} + f_{n-2},\ n > 1 \tag{12.29}$$

The point is that (12.29) can be couched in matrix terms as

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = A \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} \tag{12.30}$$

where

$$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \tag{12.31}$$

Given the initial conditions (12.28) and (12.30), we have

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = A^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{12.32}$$

In other words, our problem reduces to one of finding the powers $A, A^2, ..., A^{n-1}$.

12.4.3 Example: Matrix Inversion

Many applications make use of A^{-1} for an n x n square matrix A. In many cases, it is not computed directly, but here we address methods for direct computation.

We could use the methods of Section 12.5 to find matrix inverses, but there is alos a power series method.

Recall that for numbers x that are smaller than 1 in absolute value,

$$\frac{1}{1-x} = 1 + x + x^2 + ... \tag{12.33}$$

In algebraic terms, this would be that for an n x n matrix C,

$$(I - C)^{-1} = I + C + C^2 + ... \tag{12.34}$$

This can be shown to converge if

$$\max_{i,j} |c_{ij}| < 1 \tag{12.35}$$

To invert our matrix A, then, we can set C = I - A, giving us

$$A^{-1} = (I - C)^{-1} = I + C + C^2 + ... = I + (I - A) + (I - A)^2 + ... \tag{12.36}$$

To meet the convergence condition, we could set $\tilde{A} = dA$, where d is small enough so that (12.35) holds for $I - \tilde{A}$. This will be possible, if all the elements of A are nonnegative. We then find the inverse of dA, and in the end multiply by d to get the inverse of A.

12.4.4 Parallel Computation

So, how can we speed up the computation of matrix powers?

First, there is an important trick, useable even in the nonparallel case. Suppose we need to find only a single power, say A^{32}. We could do 31 multiplications. 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 a matrix multiplication algorithm only five times, instead of 31.

Now what if we need to find all powers, in certain range? Of course we can use the methods of Section 12.3, but actually we can do a bit better on some platforms, by exploiting the fact that both the multiplier and multiplicand are the same matrix, as follows:

Consider a situation in which we are using MPI, CUDA or **snow**, and we do matrix multiplication by some fixed tiling, say that of Section 12.3.3. Then instead of having the workers return their section of the product to the manager, just *keep those sections at the workers*. Because they are always multiplying by the same matrix, they can reuse that data, eliminating the large overhead of repeatedly shipping it back and forth between the manager and the workers.

12.5 Solving Systems of Linear Equations

Suppose we have a system of equations

$$a_{i0}x_0 + ... + a_{i,n-1}x_{n-1} = b_i, i = 0, 1, ..., n-1, \tag{12.37}$$

where the x_i are the unknowns to be solved for.

As you know, this system can be represented compactly as

$$Ax = b, \tag{12.38}$$

where A is n x n and x and b is n x 1.

12.5.1 Gaussian Elimination

Form the n x (n+1) matrix C = (A | b) by appending the column vector b to the right of A. (It may be advantageous to add padding on the right of b.)

Then we work on the rows of C, with the pseudocode for the sequential case in the most basic version being

```
1 for ii = 0 to n-1
2    divide row ii by c[i][i]
3    for r = 0 to n-1, r != i
4       replace row r by row r - c[r][ii] times row ii
```

In the divide operation in the above pseudocode, c_{ii} might be 0, or close to 0. In that case, a **pivoting** operation is performed (not shown in the pseudocode): that row is first swapped with another one further down.

This transforms C to **reduced row echelon form**, in which A is now the identity matrix I and b is now our solution vector x.

A variation is to transform only to **row echelon form**. This means that C ends up in upper triangular form, with all the elements c_{ij} with $i > j$ being 0, and with all diagonal elements being equal to 1. Here is the pseudocode:

```
1 for ii = 0 to n-1
2    divide row ii by c[i][i]
3    for r = ii+1 to n-1 // vacuous if r = n-1
```

```
    replace row r by row r - c[r][ii] times row ii
```

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]
```

12.5.2 Example: Gaussian Elimination in CUDA

Here's CUDA code for the reduced row echelon form version, suitable for a not-extremely-large matrix:

```
// linear index for matrix element at row i, column j, in an m-column
// matrix
__device__ int onedim(int i,int j,int m) {return i*m+j;}

// replace u by c* u; vector of length m
__device__ void cvec(float *u, int m, float c)
{ for (int i = 0; i < m; i++) u[i] = c * u[i]; }

// multiply the vector u of length m by the constant c (not changing u)
// and add the result to v
__device__ void vplscu(float *u, float *v, int m, float c)
{ for (int i = 0; i < m; i++) v[i] += c * u[i]; }

// copy the vector u of length m to v
__device__ void cpuv(float *u, float *v, int m)
{ for (int i = 0; i < m; i++) v[i] = u[i]; }

// solve matrix equation Ax = b; straight Gaussian elimination, no
// pivoting etc.; the matrix ab is (A|b), n rows; ab is destroyed, with
// x placed in the last column; one block, with thread i handling row i
__global__ void gauss(float *ab, int n)
{ int i,n1=n+1,abii,abme;
  extern __shared__ float iirow[];
```

```
24    int me = threadIdx.x;
25    for (i = 0; i < n; i++) {
26       if (i == me) {
27          abii = onedim(i,i,n1);
28          cvec(&ab[abii],n1-i,1/ab[abii]);
29          cpuv(&ab[abii],iirow,n1-i);
30       }
31       __syncthreads();
32       if (i != me) {
33          abme = onedim(me,i,n1);
34          vplscu(iirow,&ab[abme],n1-i,-ab[abme]);
35       }
36       __syncthreads();
37    }
38 }
```

Here we have one thread for each row, and are using just one block, so as to avoid interblock synchronization problems and to easily use shared memory. Concerning the latter, note that since the pivot row, **iirow**, is read many times, it makes sense to put it in shared memory.

Needless to say, the restriction to one block is quite significant. With a 512-thread limit per block, this would limit us to 512x512 matrices. But it's even worse than that—if shared memory is only 4K in size, in single precision that would mean something like 30x30 matrices! We could go to multiple blocks, at the cost of incurring synchronization delays coming from repeated kernel calls.

In a row echelon version of the code, we could have dynamic assignment of rows to threads, but still would eventually have load balancing issues.

12.5.3 The Jacobi Algorithm

One can rewrite (12.37) as

$$x_i = \frac{1}{a_{ii}}[b_i - (a_{i0}x_0 + ... + a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} + ... + a_{i,n-1}x_{n-1})], i = 0, 1, ..., n-1. \quad (12.39)$$

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 k^{th} iteration, we find our $(k+1)^{st}$ guess by plugging in our k^{th} guess into the right-hand side of (12.39). We keep iterating until the difference between successive guesses is small enough to indicate convergence.

This algorithm is guaranteed to converge if each diagonal element of A is larger in absolute value than the sum of the absolute values of the other elements in its row.

Parallelization of this algorithm is easy: Just assign each process to handle a section of $x = (x_0, x_1, ..., x_{n-1})$. Note that this means that each process must make sure that all other processes get the new value of its section after every iteration.

Note too that in matrix terms (12.39) can be expressed as

$$x^{(k+1)} = D^{-1}(b - Ox^{(k)}) \quad (12.40)$$

where D is the diagonal matrix consisting of the diagonal elements of A (so its inverse is just the diagonal matrix consisting of the reciprocals of those elements), O is the square matrix obtained by replacing A's diagonal elements by 0s, and $x^{(i)}$ is our guess for x in the ithiteration. This reduces the problem to one of matrix multiplication, and thus we can parallelize the Jacobi algorithm by utilizing a method for doing parallel matrix multiplication.

12.5.4 Example: OpenMP Implementation of the Jacobi Algorithm

OpenMP code for Jacobi is straightforward:

```
1  #include <omp.h>
2
3  // partitions s..e into nc chunks, placing the ith in first and last (i
4  // = 0,...,nc-1)
5  void chunker(int s, int e, int nc, int i, int *first, int *last)
6  { int chunksize = (e-s+1) / nc;
7    *first = s + i * chunksize;
8    if (i < nc-1) *last = *first + chunksize - 1;
9    else *last = e;
10 }
11
12 // returns the "dot product" of vectors u and v
13 float innerprod(float *u, float *v, int n)
14 { float sum = 0.0; int i;
15   for (i = 0; i < n; i++)
16      sum += u[i] * v[i];
17   return sum;
18 }
19
20 // solves AX = Y, A nxn; stops iteration when total change is < n*eps
21 void jacobi(float *a, float *x, float *y, int n, float eps)
22 {
23   float *oldx = malloc(n*sizeof(float));
24   float se;
25   #pragma omp parallel
26   { int i;
27      int thn = omp_get_thread_num();
28      int nth = omp_get_num_threads();
29      int first,last;
30      chunker(0,n-1,nth,thn,&first,&last);
31      for (i = first; i <= last; i++) oldx[i] = x[i] = 1.0;
32      float tmp;
33      while (1) {
34         for (i = first; i <= last; i++) {
```

```
35            tmp = innerprod(&a[n*i],oldx,n);
36            tmp -= a[n*i+i] * oldx[i];
37            x[i] = (y[i] - tmp) / a[n*i+i];
38          }
39          #pragma omp barrier
40          #pragma omp for reduction(+:se)
41          for (i = first; i <= last; i++)
42            se += abs(x[i]-oldx[i]);
43          #pragma omp barrier
44          if (se < n*eps) break;
45          for (i = first; i <= last; i++)
46            oldx[i] = x[i];
47        }
48    }
49 }
```

Note the use of the OpenMP **reduction** clause.

12.5.5 Example: R/gputools Implementation of Jacobi

Here's the R code, using **gputools**:

```
1  library(gputools)
2
3  jcb <- function(a,b,eps) {
4    n <- length(b)
5    d <- diag(a) # a vector, not a matrix
6    tmp <- diag(d) # a matrix, not a vector
7    o <- a - diag(d)
8    di <- 1/d
9    x <- b # initial guess, could be better
10   repeat {
11     oldx <- x
12     tmp <- gpumatmult(o,x)
13     tmp <- b - tmp
14     x <- di * tmp # elementwise multiplication
15     if (sum(abs(x-oldx)) < n * eps) return(x)
16   }
17 }
```

12.6 Eigenvalues and Eigenvectors

With the popularity of document search (Web search, text mining etc.), eigenanalysis has become much more broadly used. Given the size of the problems, again parallel computation is needed. This can become quite involved, with many complicated methods having been developed.

12.6.1 The Power Method

One of the simplest methods is the **power method**. Consider an nxn matrix A, with eigenvalues $\lambda_1, ..., \lambda_n$, where the labeling is such that $|\lambda_1| \geq |\lambda_2| \geq ... \geq |\lambda_n|$. We'll assume here that A is a symmetric matrix, which it is for instance in statistical applications (Section 15.4). That implies that the eigenvalues of A are real, and that the eigenvectors are orthogonal to each other.

Start with some nonzero vector x, and define the k^{th} iterate by

$$x^{(k)} = \frac{A^k x}{\| A^k x \|} \tag{12.41}$$

Under mild conditions, $x^{(k)}$ converges to an eigenvector v_1 corresponding to λ_1. Moreover, the quantities $(Ax^{(k)})'x^{(k)}$ converge to λ_1.

This method is reportedly used in Google's PageRank algorithm, which is only concerned with the largest eigenvalue/eigenvector. But what if you want more?

Consider now the matrix

$$B = A - \lambda_1 v_1 v_1' \tag{12.42}$$

where we've now scaled v_1 to have length 1.

Then

$$\begin{aligned} Bv_1 &= Av_1 - \lambda_1 v1(v_1'v_1) && (12.43)\\ &= \lambda_1 v_1 - \lambda_1 v1(1) && (12.44)\\ &= 0 && (12.45) \end{aligned}$$

and for $i > 0$,

$$\begin{aligned} Bv_i &= Av_i - \lambda_1 v1(v_1'v_i) && (12.46)\\ &= \lambda_i v_i - \lambda_1 v1(0) && (12.47)\\ &= \lambda_i v_i && (12.48) \end{aligned}$$

In other words, the eigenvalues of B are $\lambda_2, ..., \lambda_n, 0$. So we can now apply the same procedure to B to get λ_2 and v_2, and iterate for the rest.

12.6.2 Parallel Computation

To use the power method in parallel, note that this is again a situation in which we wish to compute powers of matrices. However, there is also scaling involved, as seen in (12.41). We may wish to try the "log method" of Section 12.4, with scaling done occasionally.

The CULA library for CUDA, mentioned earlier, includes routines for finding the **singular value decomposition** of a matrix, thus providing the eigenvectors.[①] The R package **gputools** has an interface to the SVD routine in CULA.

12.7 Sparse Matrices

As mentioned earlier, in many parallel processing applications of linear algebra, the matrices can be huge, even having millions of rows or columns. However, in many such cases, most of the matrix consists of 0s. In an effort to save memory, one can store such matrices in compressed form, storing only the nonzero elements.

Sparse matrices roughly fall into two categories. In the first category, the matrices all have 0s at the same known positions. For instance, in **tridiagonal** matrices, 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, such as

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 1 & 1 & 8 & 0 & 0 \\ 0 & 1 & 5 & 8 & 0 \\ 0 & 0 & 0 & 8 & 8 \\ 0 & 0 & 0 & 3 & 5 \end{pmatrix} \tag{12.49}$$

Code to deal with such matrices can then access the nonzero elements based on this knowledge.

In the second category, each matrix that our code handles will typically have its nonzero matrices in different, "random," positions. A number of methods have been developed for storing amorphous sparse matrices, such as the Compressed Sparse Row format, which we'll code in this C **struct**, representing an mxn matrix A, with k nonzero entries:

```
struct {
  int m,n; // numbers of rows and columns of A
  float *avals; // the nonzero values of A, in row-major order; length k
  int *cols; // avals[i] is in column cols[i] in A; length k
  int *rowplaces; // rowplaces[i] is the index in avals for the 1st
                  // nonzero element of row i in A (but last element
                  // is k); length m+1
}
```

For the matrix in (12.49) (if we were not to exploit its tridiagonal nature, and just treat it as amorphous):

- **m,n:** 5,5
- **avals:** 2,1,1,8,1,5,8,8,8,3,5
- **cols:** 0,0,1,2,1,2,3,3,4,3,4
- **rowplaces:** 0,2,4,6,9,11

[①] The term *singular value* is a synonym for *eigenvalue*.

For instance, look at the 4 in **rowplaces**. It's at position 2 in that array, so it says that element 4 in **avals**—the third 1—is the first nonzero element in row 2 of A. Look at the matrix, and you'll see this is true.

Parallelizing operations for sparse matrices can be done in the usual manner, e.g. breaking the rows of A into chunks. Note, though, that there could be a load-balance issue, again addressable in ways we've used before.

12.8 Libraries

Of course, remember that CUDA provides some excellent matrix-operation routines, in CUBLAS. There is also the CUSP library for sparse matrices (i.e. those with a lot of 0s). Note too the CULA library (not developed by NVIDIA, but using CUDA).

More general (i.e. non-CUDA) parallel libraries for linear algebra include ScalaPACK and PLAPACK.

第 13 章　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.

13.1 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.

13.1.1 The Separation Process

A major issue is how we separate the data into piles.

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.

```
1  int separate(int l, int h)
2  {  int ref,i,j,k,tmp;
3     ref = x[h]; i = l-1; j = h;
4     do  {
5        do i++; while (x[i] < ref && i < h);
6        do j--; while (x[j] > ref && j > l);
7        tmp = x[i];  x[i] = x[j];  x[j] = tmp;
8     } while (j > i);
9     x[j] = x[i];  x[i] = x[h];  x[h] = tmp;
10    return i;
11 }
```

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]**.

Another approach is to do a prefix scan. As an illustration, consider the array

$$28 \quad 35 \quad 12 \quad 5 \quad 13 \quad 6 \quad 8 \quad 10 \quad 168 \tag{13.1}$$

We'll take the first element, 28, as the pivot, and form a new array of 1s and 0s, where 1 means "less than the pivot":

28	35	12	5	13	6	48	10	168
0	0	1	1	1	1	0	1	0

Now form the prefix scan (Chapter 11) of that second array, with respect to addition. It will be an *exclusive* scan (Section 11.3). This gives us

28	35	12	5	13	6	48	10	168
0	0	1	1	1	1	0	1	0
0	0	0	1	2	3	3	4	4

Now, the key point is that for every element 1 in that second row, the corresponding element in the third row shows where the first-row element should be placed under the separation operation! Here's why:

The elements 12, 5, 13, 6 and 10 should go in the first pile, which in an in-place separation would means indices 0, 1, 2, 3, and 4. Well, as you can see above, these are precisely the values shown in the third row for 12, 5, 13, 6 and 10, all of which have 1s in the second row.

The pivot, 28, then should immediately follow that low pile, i.e. it should be placed at index 5.

We can simply place the high pile at the remaining indicies, 6 through 8 (though we'll do it more systematically below).

In general for an array of length k, we:

- form the second row of 1s and 0s indicating < pivot
- form the third row, the exclusive prefix scan
- for each 1 in the second row, place the corresponding element in row 1 into the spot indicated by row 3
- place the pivot in the place indicated by 1 plus m, the largest value in row 3
- form row 4, equal to (0,1,...,k-1) minus row 3 plus m
- for each 0 in the second row, place the corresponding element in row 1 into the spot indicated by row 4

Note that this operation, using scan, could be used an an alternative to the **separate()** function above. But it could be done in parallel; more on this below.

13.1.2 Example: OpenMP 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.

Recall that another implementation, using the **task** directive, was given earlier in Section 4.5.

In both of these implementations, we used the function **separate()** defined above. So, different threads apply different separation operations to different subarrays. An alternative would be to place the parallelism in the separation operation itself, using the parallel algorithms for prefix scan in Chapter 11.

13.1.3 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 7 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.

13.2 Mergesorts

13.2.1 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 13.2.3.

13.2.2 Shared-Memory Mergesort

This is similar to the patterns for shared-memory quicksort in Section 13.1.2 above.

13.2.3 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
```

There is quite a load balancing issue here. On the one hand, 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. Otherwise, "upstream" nodes will frequently have no work to do.

On the other hand, the larger the chunk size, the earlier the leaf nodes will have no work to do. So for any particular platform, there will be some optimal chunk size, which would need to be determined by experimentation.

13.2.4 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
```

13.2.5 Bitonic Mergesort

Definition: A sequence $(a_0, a_1, .., a_{k-1})$ is called **bitonic** if either of the following conditions holds:

(a) The sequence is first nondecreasing then nonincreasing, meaning that for some r

$$(a_0 \leq a_1 \leq ... \leq a_r \geq a_{r+1} \geq a_{n-1})$$

(b) The sequence can be converted to the form in (a) by rotation, i.e. by moving the last k elements from the right end to the left end, for some k.

As an example of (b), 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). Or we could just rotate by one element, moving the 2 to forming (2,3,8,12,15,14,5,1).

Note that the definition includes the cases in which the sequence is purely nondecreasing (r = n-1) or purely nonincreasing (r = 0).

Also included are "V-shape" sequences, in which the numbers first decrease then increase, such as (12,5,2,8,20). By (b), these can be rotated to form (a), with (12,5,2,8,20) being rotated to form (2,8,20,12,5), an "A-shape" sequence.

(For convenience, from here on I will use the terms *increasing* and *decreasing* instead of *nonincreasing* and *nondecreasing*.)

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);
   }
}
```

This can be parallelized in the same ways we saw for Quicksort earlier.

So much for sorting bitonic sequences. But what about general sequences?

We can proceed as follows, using our function **sortbitonic()** above:

1. For each i = 0,2,4,...,n-2:
   - Each of the pairs (a_i, a_{i+1}), i = 0,2,...,n-2 is bitonic, since *any* 2-element array is bitonic!
   - Apply **sortbitonic()** to (a_i, a_{i+1}). In this case, we are simply doing a compare-exchange.
   - 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.
2. 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.
3. 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.

13.3 The Bubble Sort and Its Cousins

13.3.1 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 13.2.4 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 13.2.4.

13.3.2 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.

13.3.3 Example: CUDA Implementation of Odd/Even Transposition Sort

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

// compare and swap; copies from the f to t, swapping f[i] and
// f[j] if the higher-index value is smaller; it is required that i < j
__device__ void cas(int *f,int *t,int i,int j, int n, int me)
{
   if (i < 0 || j >= n) return;
   if (me == i) {
      if (f[i] > f[j]) t[me] = f[j];
      else t[me] = f[i];
   } else {  // me == j
      if (f[i] > f[j]) t[me] = f[i];
      else t[me] = f[j];
   }
}
```

```
18
19 // does one iteration of the sort
20 __global__ void oekern(int *da, int *daaux, int n, int iter)
21 {  int bix = blockIdx.x;  // block number within grid
22    if (iter % 2) {
23       if (bix % 2) cas(da,daaux,bix-1,bix,n,bix);
24       else cas(da,daaux,bix,bix+1,n,bix);
25    } else {
26       if (bix % 2) cas(da,daaux,bix,bix+1,n,bix);
27       else cas(da,daaux,bix-1,bix,n,bix);
28    }
29 }
30
31 // sorts the array ha, length n, using odd/even transp. sort;
32 // kept simple for illustration, no optimization
33 void oddeven(int *ha, int n)
34 {
35    int *da;
36    int dasize = n * sizeof(int);
37    cudaMalloc((void **)&da,dasize);
38    cudaMemcpy(da,ha,dasize,cudaMemcpyHostToDevice);
39    // the array daaux will serve as "scratch space"
40    int *daaux;
41    cudaMalloc((void **)&daaux,dasize);
42    dim3 dimGrid(n,1);
43    dim3 dimBlock(1,1,1);
44    int *tmp;
45    for (int iter = 1; iter <= n; iter++) {
46       oekern<<<dimGrid,dimBlock>>>(da,daaux,n,iter);
47       cudaThreadSynchronize();
48       if (iter < n) {
49          // swap pointers
50          tmp = da;
51          da = daaux;
52          daaux = tmp;
53       } else
54          cudaMemcpy(ha,daaux,dasize,cudaMemcpyDeviceToHost);
55    }
56 }
```

Recall that in CUDA code, separate blocks of threads cannot synchronize with each other. Unless we deal with just a single block, this necessitates limiting the kernel to a single iteration of the algorithm, so that as iterations progress, execution alternates between the device and the host.

Moreover, we do not take advantage of shared memory. One possible solution would be to use **___syncthreads()** within each block for most of the compare-and-exchange operations, and then having the host take care of the operations on the boundaries between blocks.

13.4 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.

13.5 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.

OpenMP code for this was given in Section 1.3.2. Here is similar MPI code below (various improvements could be made, e.g. with broadcast):

```
// bucket sort, bin boundaries known in advance

// node 0 is manager, all else worker nodes; node 0 sends full data, bin
// boundaries to all worker nodes; i-th worker node extracts data for
```

```
// bin i-1, sorts it, sends sorted chunk back to node 0; node 0 places
// sorted results back in original array

// not claimed efficient; e.g. could be better to have manager place
// items into bins

#include <mpi.h>

#define MAX_N 100000 // max size of original data array
#define MAX_NPROCS 100 // max number of MPI processes
#define DATA_MSG 0 // manager sending original data
#define BDRIES_MSG 0 // manager sending bin boundaries
#define CHUNKS_MSG 2 // workers sending their sorted chunks

int nnodes, //
    n, // size of full array
    me, // my node number
    fulldata[MAX_N],
    tmp[MAX_N],
    nbdries, // number of bin boundaries
    counts[MAX_NPROCS];
float bdries[MAX_NPROCS-2]; // bin boundaries

int debug,debugme;

init(int argc, char **argv)
{
   int i;
   debug = atoi(argv[3]);
   debugme = atoi(argv[4]);
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&nnodes);
   MPI_Comm_rank(MPI_COMM_WORLD,&me);
   nbdries = nnodes - 2;
   n = atoi(argv[1]);
   int k = atoi(argv[2]); // for random # gen
   // generate random data for test purposes
   for (i = 0; i < n; i++) fulldata[i] = rand() % k;
   // generate bin boundaries for test purposes
   for (i = 0; i < nbdries; i++) {
      bdries[i] = i * (k+1) / ((float) nnodes);
   }
}
```

```
48
49 void managernode()
50 {
51    MPI_Status status;
52    int i;
53    int lenchunk; // length of a chunk received from a worker
54    // send full data, bin boundaries to workers
55    for (i = 1; i < nnodes; i++) {
56       MPI_Send(fulldata,n,MPI_INT,i,DATA_MSG,MPI_COMM_WORLD);
57       MPI_Send(bdries,nbdries,MPI_FLOAT,i,BDRIES_MSG,MPI_COMM_WORLD);
58    }
59    // collect sorted chunks from workers, place them in their proper
60    // positions within the original array
61    int currposition = 0;
62    for (i = 1; i < nnodes; i++) {
63       MPI_Recv(tmp,MAX_N,MPI_INT,i,CHUNKS_MSG,MPI_COMM_WORLD,&status);
64       MPI_Get_count(&status,MPI_INT,&lenchunk);
65       memcpy(fulldata+currposition,tmp,lenchunk*sizeof(int));
66       currposition += lenchunk;
67    }
68    if (n < 25) {
69       for (i = 0; i < n; i++) printf("%d ",fulldata[i]);
70       printf("\n");
71    }
72 }
73
74 // adds xi to the part array, increments npart, the length of part
75 void grab(int xi, int *part, int *npart)
76 {
77     part[*npart] = xi;
78     *npart += 1;
79 }
80
81 int cmpints(int *u, int *v)
82 { if (*u < *v) return -1;
83    if (*u > *v) return 1;
84    return 0;
85 }
86
87 void getandsortmychunk(int *tmp, int n, int *chunk, int *lenchunk)
88 {
89    int i,count = 0;
90    int workernumber = me - 1;
```

```
91      if (me == debugme) while (debug) ;
92    for (i = 0; i < n; i++) {
93       if (workernumber == 0) {
94          if (tmp[i] <= bdries[0]) grab(tmp[i],chunk,&count);
95       }
96       else if (workernumber < nbdries-1) {
97          if (tmp[i] > bdries[workernumber-1] &&
98              tmp[i] <= bdries[workernumber]) grab(tmp[i],chunk,&count);
99       } else
100         if (tmp[i] > bdries[nbdries-1]) grab(tmp[i],chunk,&count);
101   }
102   qsort(chunk,count,sizeof(int),cmpints);
103   *lenchunk = count;
104 }
105
106 void workernode()
107 {
108   int n,fulldata[MAX_N], // size and storage of full data
109       chunk[MAX_N],
110       lenchunk,
111       nbdries; // number of bin boundaries
112   float bdries[MAX_NPROCS-1]; // bin boundaries
113   MPI_Status status;
114   MPI_Recv(fulldata,MAX_N,MPI_INT,0,DATA_MSG,MPI_COMM_WORLD,&status);
115   MPI_Get_count(&status,MPI_INT,&n);
116   MPI_Recv(bdries,MAX_NPROCS-2,MPI_FLOAT,0,BDRIES_MSG,MPI_COMM_WORLD,&status)
          ;
117   MPI_Get_count(&status,MPI_FLOAT,&nbdries);
118   getandsortmychunk(fulldata,n,chunk,&lenchunk);
119   MPI_Send(chunk,lenchunk,MPI_INT,0,CHUNKS_MSG,MPI_COMM_WORLD);
120 }
121
122 int main(int argc,char **argv)
123 {
124   int i;
125   init(argc,argv);
126   if (me == 0) managernode();
127   else workernode();
128   MPI_Finalize();
129 }
```

13.6 Radix Sort

The radix sort is essentially a special case of a bucket sort. If we have 16 threads, say, we could determine a datum's bucket by its lower 4 bits. As long as our data is uniformly distributed under the mod 16 operation, we would not need to do any sampling.

The CUDPP GPU library uses a radix sort. The buckets are formed one bit at a time, using segmented scan as above.

13.7 Enumeration Sort

This one is really simple. Take for instance the array (12,5,13,18,6). There are 2 elements less than 12, so in the end, it should go in position 2 of the sorted array, (5,6,12,13,18).

Say we wish to sort $\mathbf{x}$, which for convenience we assume contains no tied values. Then the pseudocode for this algorithm, placing the results in $\mathbf{y}$, is

```
for all i in 0...n-1:
   count = 0
   elt = x[i]
   for all j in 0...n-1:
      if x[j] < elt then count++
   y[count] = elt
```

The outer (or inner) loop is easily parallelized.

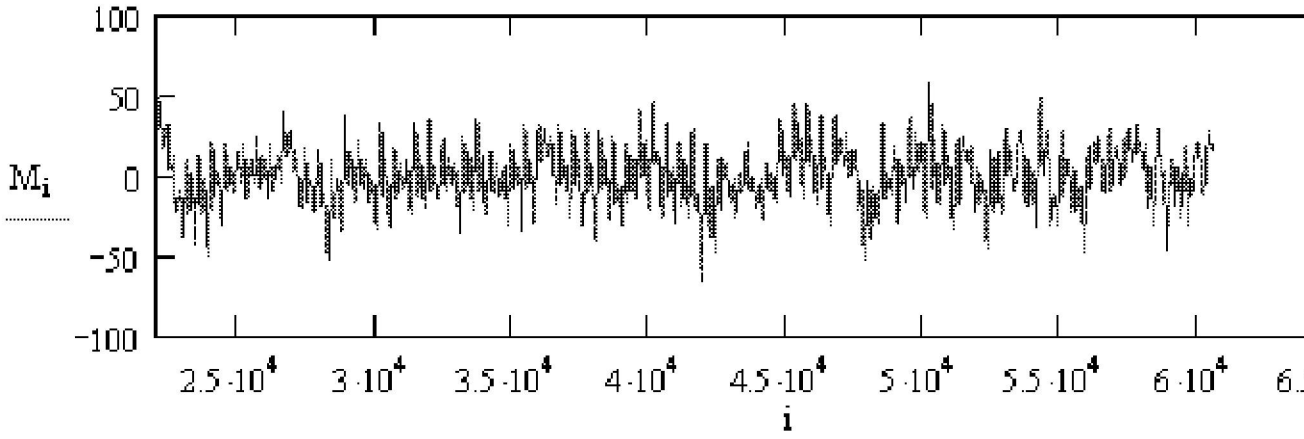
第 14 章　Parallel Computation for Audio and Image Processing

Mathematical computations involving images can become quite intensive, and thus parallel methods are of great interest. Here we will be primarily interested in methods involving **Fourier** analysis.

14.1 General Principles

14.1.1 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]



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,

$$f_0 = \frac{1}{T} \tag{14.1}$$

Recall also from calculus that we can write a function g(t) (not necessarily periodic) as a Taylor series, which is an "infinite polynomial":

$$g(t) = \sum_{n=0}^{\infty} c_n t^n. \tag{14.2}$$

[①]Reproduced here by permission of Prof. Peter Hamburger, Indiana-Purdue University, Fort Wayne. See http://www.ipfw.edu/math/Workshop/PBC.html

The specific values of the c_n may be derived by differentiating both sides of (14.2) and evaluating at t = 0, yielding

$$c_n = \frac{g^{(n)}(0)}{n!}, \tag{14.3}$$

where $g^{(j)}$ denotes the ith derivative of g().

For instance, for e^t,

$$e^t = \sum_{n=0}^{\infty} \frac{1}{n!} t^n \tag{14.4}$$

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

$$g(t) = \sum_{n=0}^{\infty} a_n \cos(2\pi n f_0 t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n f_0 t) \tag{14.5}$$

for some set of weights a_n and b_n. Here, instead of having a weighted sum of terms

$$1,\ t,\ t^2,\ t^3,\ ... \tag{14.6}$$

as in a Taylor series, we have a weighted sum of terms

$$1,\ \cos(2\pi f_0 t),\ \cos(4\pi f_0 t),\ \cos(6\pi f_0 t),\ ... \tag{14.7}$$

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]

$$a_0 = \frac{1}{T} \int_0^T g(t)\ dt \tag{14.8}$$

$$a_n = \frac{2}{T} \int_0^T g(t)\ cos(2\pi n f_0 t)\ dt \tag{14.9}$$

$$b_n = \frac{2}{T} \int_0^T g(t)\ sin(2\pi n f_0 t)\ dt \tag{14.10}$$

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.

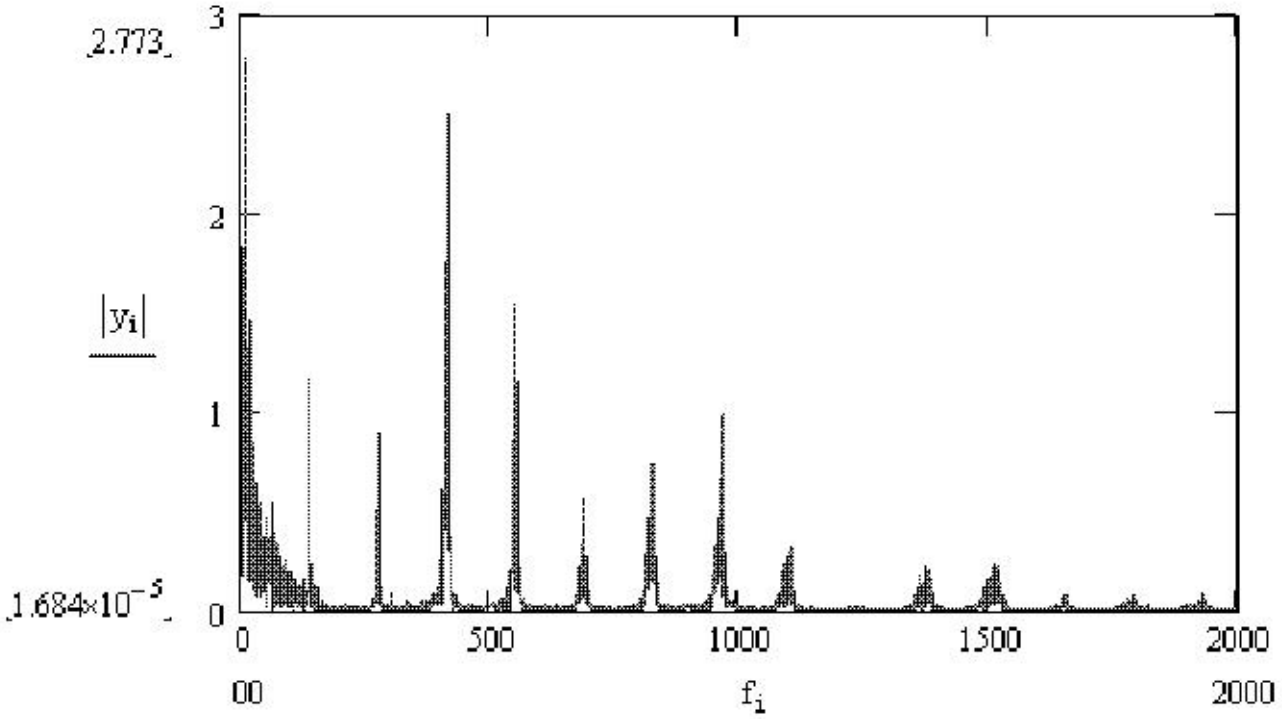
Since g(t) is a graph of loudness against time, this representation of the sound is called the **time domain**. When we find the Fourier series of the sound, the set of weights a_n and b_n is

[②] The get an idea as to how these formulas arise, see Section 14.9. But for now, if you integrate both sides of (14.5), you will at least verify that the formulas below do work.

said to be a representation of 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 (14.8), (14.9), (14.10) and (14.5).

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

$$e^{i\theta} = \cos(\theta) + i \ \sin(\theta), \tag{14.11}$$

where $i = \sqrt{-1}$.[③]

The complex form of (14.5) is

$$g(t) = \sum_{j=-\infty}^{\infty} c_j e^{2\pi i j \frac{t}{T}}. \tag{14.12}$$

The c_j are now generally complex numbers. They are functions of the a_j and b_j, and thus form the frequency spectrum.

Equation (14.12) has a simpler, more compact form than (14.5). Do you now see why I referred to Fourier series as trig polynomials? The series (14.12) involves the j^{th} powers of $e^{2\pi \frac{t}{T}}$.

[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 (14.12), being a case in point.

It is not assumed that you know complex variables well. All that is required is knowledge of how to add, subtract, multiply and divide, and the definition of $|c|$ for complex c.

14.1.2 Two-Dimensional Fourier Series

Let's now move from sounds to images. Just as we were taking time to be a continuous variable above, for the time being we are taking the position within an image to be continuous too; this is equivalent to having infinitely many pixels. Here g() is a function of two variables, g(u,v), where u and v are the horizontal and vertical coordinates of a point in the image, with g(u,v) being the intensity of the image at that point. If it is a gray-scale image, the intensity is whiteness of the image at that point, 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.

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.

14.2 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.

Integrals like (14.8) now change to sums.

14.2.1 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:

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j e^{-2\pi ijk/n} = \frac{1}{n} \sum_{j=0}^{n-1} x_j q^{jk} \tag{14.13}$$

where

$$q = e^{-2\pi i/n} \tag{14.14}$$

again with $i = \sqrt{-1}$. The array C of complex numbers c_k is called the **discrete Fourier transform** (DFT) of X. Note that (14.13) is basically a discrete analog of (14.9) and (14.10).

Note that instead of having infinitely many frequencies, we only have n of them, i.e. the n original data points x_j map to n frequency weights c_k.⑤

The quantity q is a n^{th} root of 1:

④See Section 14.10 for the reasons behind this.

⑤Actually, in the case of x_j real, which occurs with sound data, we really get only n/2 frequencies. The weight of the frequences after k = n/2 turn out to be the **conjugates** of those before n/2, where the conjugate of a+bi is defined to be a-bi.

$$q^n = e^{-2\pi i} = \cos(-2\pi) + i\sin(-2\pi) = 1 \tag{14.15}$$

Equation (14.13) can be written as

$$C = \frac{1}{n}AX, \tag{14.16}$$

where X is the vector x_j and

$$A = \begin{pmatrix} 1 & 1 & 1 & ... & 1 \\ 1 & q & q^2 & ... & q^{n-1} \\ ... & ... & ... & ... & ... \\ 1 & q^{n-1} & q^{2(n-1)} & ... & q^{(n-1)(n-1)} \end{pmatrix} \tag{14.17}$$

Here's R code to calculate A:

```
makeamat <- function(n,u)  {
   m <- matrix(nrow=n,ncol=n)
   for (i in 1:n) {
      for (j in i:n) {
         if (i == j) {
            m[i,i] <- u^((i-1)^2)
         }
         else {
            m[i,j] <- u^((i-1)*(j-1))
            m[j,i] <- m[i,j]
         }
      }
   }
   m
}
```

14.2.2 Inversion

As in the continuous case, the DFT is a one-to-one transformation. so we can recover each domain from the other. The details are important:

The matrix A in (14.17) is a special case of **Vandermonde** matrices, known to be invertible. In fact, if we think of that matrix as a function of q, A(q), then it turns out that

$$[A(q)]^{-1} = \frac{1}{n}A(\frac{1}{q}) \tag{14.18}$$

Thus (14.16) becomes

$$X = n[A(q)]^{-1}C = A(\frac{1}{q})C \tag{14.19}$$

In nonmatrix terms:

$$x_j = \sum_{k=0}^{n-1} c_k e^{2\pi ijk/n} = \sum_{k=0}^{n-1} c_k q^{-jk} \tag{14.20}$$

Equation (14.20) is basically a discrete analog of (14.5).

Alternate Formulation

Equation (14.16) has a factor 1/n while (14.19) doesn't. In order to achieve symmetry, some authors of material on DFT opt to define the DFT and its inverse with $1/\sqrt{n}$ in (14.13) instead of 1/n, and by adding a factor $1/\sqrt{n}$ in (14.20). They then include a factor $1/\sqrt{n}$ in (14.17), with the result that $[A(q)]^{-1} = A(1/q)$. Thus everything simplifies.

Other formulations are possible. For instance, the R **fft()** routine's documentation says it's "unnormalized," meaning that there is neither a $1/n$ nor a $1/\sqrt{n}$ in (14.20). When using a DFT routine, be sure to determine what it assumes about these constant factors.

14.2.3 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 (14.13) becomes

$$c_{rs} = \frac{1}{n}\frac{1}{m}\sum_{j=0}^{n-1}\sum_{k=0}^{m-1} x_{jk} e^{-2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{14.21}$$

where r = 0,1,...,n-1, s = 0,1,...,m-1.

Its inverse is

$$x_{rs} = \sum_{j=0}^{n-1}\sum_{k=0}^{m-1} c_{jk} e^{2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{14.22}$$

14.3 Parallel Computation of Discrete Fourier Transforms

14.3.1 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 (14.13) can be rewritten as

$$c_k = \frac{1}{n}\left[\sum_{j=0}^{m-1} x_{2j} q^{2jk} + \sum_{j=0}^{m-1} x_{2j+1} q^{(2j+1)k},\right] \tag{14.23}$$

where $m = n/2$.

After some algebraic manipulation, this becomes

$$c_k = \frac{1}{2}\left[\frac{1}{m}\sum_{j=0}^{m-1} x_{2j} z^{jk} + q^k \frac{1}{m}\sum_{j=0}^{m-1} x_{2j+1} z^{jk}\right] \tag{14.24}$$

where $z = e^{-2\pi i/m}$.

A look at Equation (14.24) shows that the two sums within the brackets have the same form as Equation (14.13). In other words, Equation (14.24) 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 13.

In a message-passing setting, again because this is a divide-and-conquer algorithm, we can use the pattern of Hyperquicksort, also in Chapter 13.

Some digital signal processing chips implement this in hardware, with a special interconnection network to implement this algorithm.

14.3.2 A Matrix Approach

The matrix form of (14.13) is

$$C = \frac{1}{n} AX \tag{14.25}$$

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, as in Chapter 12.

Divide-and-conquer tends not to work too well in shared-memory settings, because after some point, fewer and fewer threads will have work to do. Thus this matrix formulation is quite valuable.

14.3.3 Parallelizing Computation of the Inverse Transform

The form of the DFT (14.13) and its inverse (14.20) are very similar. For example, the inverse transform is again of a matrix form as in (14.25); 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.

14.3.4 Parallelizing Computation of the Two-Dimensional Transform

Regroup (14.21) as:

$$c_{rs} = \frac{1}{n} \sum_{j=0}^{n-1} \left(\frac{1}{m} \sum_{k=0}^{m-1} x_{jk} e^{-2\pi i (\frac{ks}{m})} \right) e^{-2\pi i (\frac{jr}{n})} \tag{14.26}$$

$$= \frac{1}{n} \sum_{j=0}^{n-1} y_{js} e^{-2\pi i (\frac{jr}{n})} \tag{14.27}$$

Note that y_{js}, i.e. the expression between the large parentheses, is the s^{th} component of the DFT of the j^{th} row of our data. And hey, the last expression (14.27) above is in the same form as (14.13)! 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**.

[6] In fact, one can obtain the new matrix easily from the old, as explained in Section 14.9.

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.

14.4 Available FFT Software

14.4.1 R

As of now, R only offers serial computation, through its function **fft()**. It works on both one- and two-dimensional (or more) data. If its argument **inverse** is set to TRUE, it will find the inverse.

Parallel computation of a two-dimensional transform can be easily accomplished by using **fft()** together with the approach in Section 14.3.4 and one of the packages for parallel R in Chapter 10. Here's how to do it in **snow**:

```
1 parfft2 <- function(cls,m) {
2   tmp <- parApply(cls,m,1,fft)
3   parApply(cls,tmp,1,fft)
4 }
```

Recall that when **parApply()** is called with a vector-valued function argument, the output from row i of the input matrix is placed in *column* i of the output matrix. Thus in the second call above, we used rows (argument 1) instead of columns.

14.4.2 CUFFT

Remember that CUDA includes some excellent FFT routines, in CUFFT.

14.4.3 FFTW

FFTW ("Fastest Fourier Transform in the West") is available for free download at `http://www.fftw.org`. It includes versions callable from OpenMP and MPI.

14.5 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.

14.5.1 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

[7] Remember, there may be three intensity values per pixel, for red, green and blue.

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 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.

14.5.2 Example: Audio Smoothing in R

Below is code to do smoothing on sound. It inputs a sound sequence **snd**, and performs low-pass filtering, setting to 0 all DFT terms having k greater than `maxidx` in (14.13).

```
p <- function(snd,maxidx) {
   four <- fft(snd)
   n <- length(four)
   newfour <- c(four[1:maxidx],rep(0,n-maxidx))
   return(Re(fft(newfour,inverse=T)/n))
}
```

Here the **Re()** function extracts the real part of a complex number.

14.5.3 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.

Again, this means first taking the Fourier transform of the original, then deleting the low-frequency terms, then taking the inverse transform to go back to the spatial domain.

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`, including the original pictures, which may not show up well in our printed book here.

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.

14.6 R Access to Sound and Image Files

In order to apply these transformations to sound and image files, you need to extract the actual data from the files. The formats are usually pretty complex. You can do this easily using the R **tuneR** and **pixmap** libraries.

After extracting the data, you can apply the transformations, then transform back to the time/spatial domain, and replace the data component of the original class.

14.7 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.

14.8 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.

14.9 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 (14.13).

Define X and C as in Section 14.2. 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

$$\overline{e^{i\theta}} = \cos\theta - i\sin\theta == \cos(-\theta) + i\sin(-\theta) = e^{-i\theta} \tag{14.28}$$

Define an inner product ("dot product"),

$$[u, w] = \frac{1}{n}\sum_{j=0}^{n-1} u_j \bar{w}_j. \tag{14.29}$$

Define

$$v_h = (1, q^{-h}, q^{-2h}, ..., q^{-(n-1)h}), h = 0, 1, ..., n-1. \tag{14.30}$$

Then it turns out that the v_h form an orthonormal basis for V.[11] For example, to show orthnogonality, observe that for $r \neq s$

[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.

[11] Recall that this means that these vectors are orthogonal to each other, and have length 1, and that they span V.

$$[v_r, v_s] = \frac{1}{n}\sum_{j=0}^{n-1} v_{rj}\overline{v_{sj}} \tag{14.31}$$

$$= \frac{1}{n}\sum_{j=0} q^{j(-r+s)} \tag{14.32}$$

$$= \frac{1-q^{(-r+s)n}}{n(1-q)} \tag{14.33}$$

$$= 0, \tag{14.34}$$

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 (14.13).

The fact that we have an orthonormal basis for V here means that the matrix A/n in (14.25) 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 (14.25), we can easily get back to X from C, via

$$X = BC = \frac{1}{n}\bar{A}^t C. \tag{14.35}$$

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 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.

14.10 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

[12] And in fact will probably be deliberately filtered out.

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 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.

第 15 章　Parallel Computation in Statistics/Data Mining

How did the word *statistics* get supplanted by *data mining*? In a word, it is a matter of scale.

In the old days of statistics, a data set of 300 observations on 3 or 4 variables was considered large. Today, the widespread use of computers and the Web yield data sets with numbers of observations that are easily in the tens of thousands range, and in a number of cases even tens of millions. The numbers of variables can also be in the thousands or more.

In addition, the methods have become much more combinatorial in nature. In a classification problem, for instance, the old discriminant analysis involved only matrix computation, whereas a nearest-neighbor analysis requires far more computer cycles to complete.

In short, this calls for parallel methods of computation.

15.1 Itemset Analysis

15.1.1 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, as mentioned earlier.

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.[①]

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**:

15.1.2 The Market Basket Problem

Consider an online bookstore that 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 i^{th} sale included book j, i = 0,1,...,s-1, j = 0,1,...,t-1. So each row of S represents one

[①] 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. It's a good idea, but overfitting can still occur even with this precaution.

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** $I \to J$ is simply an ordered pair of disjoint itemsets I and J.
- The **support** of an an association rule $I \to J$ is the proportion of records which include both I and J.
- The **confidence** of an association rule $I \to J$ 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 confidence is high in the book example, 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 may not be 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.

15.1.3 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. In the online bookstore, for instance, this would mean finding all individual books that appear in at least r of our sales transaction records, where r is our threshold.

At the second level, we find the frequent 2-item itemsets, e.g. all pairs of books that appear in at least r sales records, 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.

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 ∪ K
        if support(Q) exceeds support threshold
          add Q to F_i
  if F_i is empty break
return ∪_i F_i
```

In other words, we are building up the itemsets of size i from those of size i-1, adding all possible choices of one element to each of the latter.

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.

15.1.4 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, say in the message-passing case.

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.

15.2 Probability Density Estimation

Let X denote some quantity of interest in a given population, say people's heights. Technically, the **probability density function** of X, typically denoted by f, is a function on the real line with the following properties:

- $f(t) \geq 0$ for all t
- for any $r < s$,

$$P(r < X < s) = \int_r^s f(t)\ dt \tag{15.1}$$

(Note that this implies that f integrates to 1.)

This seems abstract, but it's really very simple: Say we have data on X, n sample values $X_1, ..., X_n$, and we plot a histogram from this data. Then *f is what the histogram is estimating.* If we have more and more data, the histogram gets closer and closer to the true f.[②]

So, how do we estimate f, and how do we use parallel computing to reduce the time needed?

[②] The histogram must be scaled to have total area 1. Most statistical programs have options for this.

15.2.1 Kernel-Based Density Estimation

Histogram computation breaks the real down into intervals, and then counts how many X_i fall into each interval. This is fine as a crude method, but one can do better.

No matter what the interval width is, the histogram will consist of a bunch of rectanges, rather than a smooth curve. This problem basically stems from a lack of weighting on the data.

For example, suppose we are estimating f(25.8), and suppose our histogram interval is [24.0,26.0], with 54 points falling into that interval. Intuitively, we can do better if we give the points closer to 25.8 more weight.

One way to do this is called **kernel-based** density estimation, which for instance in R is handled by the function **density()**.

We need a set of weights, more precisely a weight function k, called the **kernel**. Any nonnegative function which integrates to 1—i.e. a density function in its own right—will work. Typically k is taken to be the Gaussian or normal density function,

$$k(u) = \frac{1}{\sqrt{2\pi}} e^{-0.5u^2} \tag{15.2}$$

Our estimator is then

$$\widehat{f}(t) = \frac{1}{nh} \sum_{i=1}^{n} k\left(\frac{t - X_i}{h}\right) \tag{15.3}$$

In statistics, it is customary to use the ^symbol (pronounced "hat") to mean "estimate of." Here $\widehat{f}$ means the estimate of f.

Note carefully that we are estimating an entire function! There are infinitely many possible values of t, thus infinitely many values of f(t) to be estimated. This is reflected in (15.3), as $\widehat{f}(t)$ does indeed give a (potentially) different value for each t.

Here h, called the *bandwidth*, is playing a role analogous to the interval width in the case of histograms. We must choose the value of h, just like for a histogram we must choose the bin width.[③]

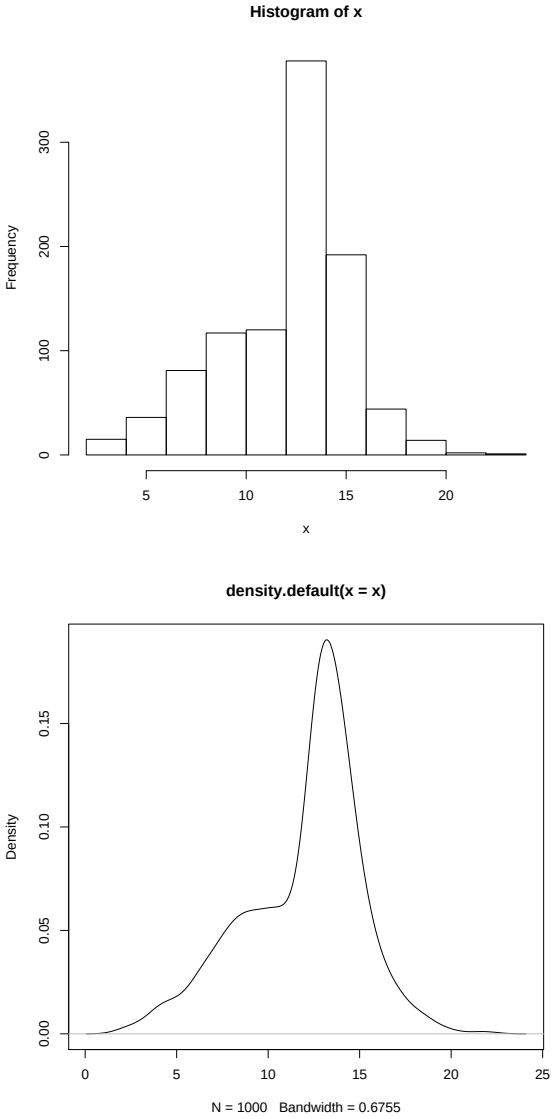
Again, this looks very abstract, but all it is doing is assigning weights to the data. Consider our example above in which we wish to estimate f(25.8), i.e. t = 25.8 and suppose we choose h to be 6.0. If say, X_{88} is 1209.1, very far as away from 25.8, we don't want this data point to have much weight in our estimation of f(25.8). Well, it won't have much weight at all, because the quantity

$$u = \frac{25.8 - 88}{6} \tag{15.4}$$

will be very large, and (15.2) will be tiny, as u will be way, way out in the left tail.

Now, keep all this in perspective. In the end, we will be plotting a curve, *just like we do with a histogram*. We simply have a more sophiticated way to do this than plotting a histogram. Following are the graphs generated first by the histogram method, then by the kernel method, on the same data:

[③] Some statistical programs will choose default values, based on theory.



There are many ways to parallelize this computation, such as:

- Remember, we are going to compute (15.3) for many values of t. So, we can just have each process compute a block of those values.
- We may wish to try several different values of h, just as we might try several different interval widths for a histogram. We could have each process compute using its own values of h.
- It can be shown that (15.3) has the form of something called a **convolution**. The theory of convolution would take us too far afield,④ but this fact is useful here, as the Fourier transform

④

If you've seen the term before and are curious as to how this is a convolution, read on:

Write (15.3) as

$$\widehat{f}(t) = \sum_{i=1}^{n} \frac{1}{h} k\left(\frac{t - X_i}{h}\right) \cdot \frac{1}{n} \tag{15.5}$$

Now consider two artificial random variables U and V, created just for the purpose of facilitating computation, defined as follows.

The random variable U takes on the values ih with probability $g \cdot \frac{1}{h} k(i)$, i = -c,-c+1,...,0,1,...,c for some value of c that we choose to cover most of the area under k, with g chosen so that the probabilities sum to 1. The random variable V takes on the values $X_1, ..., X_n$ (considered fixed here), with probability 1/n each. U and V are set to be independent.

Then (g times) (15.5) becomes P(U+V=t), exactly what convolution is about, the probability mass function (or density, in the continuous case) of a random variable arising as the sum of two independent nonnegative random variables.

of a convolution can be shown to be the product of the Fourier transforms of the two convolved components.[5] In other words, *this reduces the problem to that of parallelizing Fourier transforms*—something we know how to do, from Chapter 14.

15.2.2 Histogram Computation for Images

In image processing, histograms are used to find tallies of how many pixels there are of each intensity. (Note that there is thus no interval width issue, as there is a separate "interval" value for each possible intensity level.) The serial pseudocode is:

```
for i = 1,...,numintenslevels:
   count = 0
   for row = 1,...,numrows:
      for col = 1,...,numcols:
         if image[row][col] == i: count++
   hist[i] = count
```

On the surface, this is certainly an "embarrassingly parallel" problem. In OpenMP, for instance, we might have each thread handle a block of rows of the image, i.e. parallelize the **for row** loop. In CUDA, we might have each thread handle an individual pixel, thus parallelizing the nested **for row/col** loops.

However, to make this go fast is a challenge, say in CUDA, due to issues of what to store in shared memory, when to swap it out, etc. A very nice account of fine-tuning this computation in CUDA is given in *Histogram Calculation in CUDA*, by Victor Podlozhnyuk of NVIDIA, 2007, `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf`. The actual code is at `http://developer.download.nvidia.com/compute/cuda/sdk/website/Data-Parallel_Algorithms.html#histogram`. A summary follows:

(Much of the research into understand Podlozhnyuk's algorithm was done by UC Davis graduate student Spencer Mathews.)

Podlozhnyuk's overall plan is to have the threads compute subhistograms for various chunks of the image, then merge the subhistograms to create the histogram for the entire data set. Each thread will handle 1/k of the image's pixels, where k is the total number of threads in the grid, i.e. across all blocks.

In Podlozhnyuk's first cut at the problem, he maintains a separate subhistogram for each thread. He calls this version of the code **histogram64**. The name stems from the fact that only 64 intensity levels are used, i.e. the more significant 6 bits of each pixel's data byte. The reason for this restriction will be discussed later.

Each thread will store its subhistogram as an array of bytes; the count of pixels that a thread finds to have intensity i will be stored in the i^{th} byte of this array. Considering the content of a byte as an unsigned number, that means that each thread can process only 255 pixels.

The subhistograms will be stored together in a two-dimensional array, the j^{th} being the subhistogram for thread j. Since the subhistograms are accessed repeatedly, we want to store this two-dimensional array in shared memory. (Since each pixel will be read only once, there would be no value in storing it in shared memory, so it is in global memory.)

[5] Again, if you have some background in probability and have see characteristic functions, this fact comes from the fact that the characteristic function of the sum of two independent random variables is equal to the product of the characteristic functions of the two variables.

The main concern is bank conflicts. As the various threads in a block write to the two-dimensional array, they may collide with each other, i.e. try to write to different locations within the same bank. But Podlozhnyuk devised a clever way to stagger the accesses, so that in fact there are no bank conflicts at all.

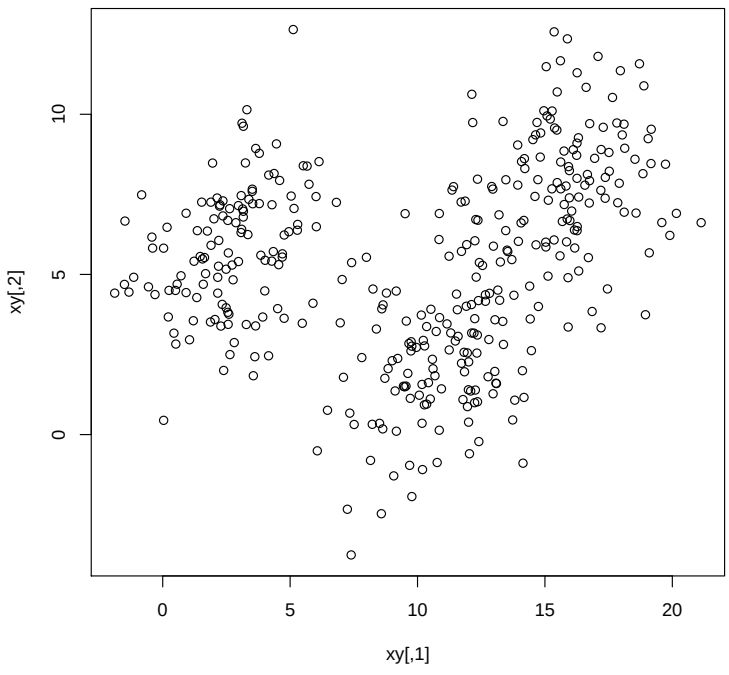
In the end, the many subhistograms within a block must be merged, and those merged counts must in turn be merged across all blocks. The former operation is done again by careful ordering to avoid any bank conflicts, and then the latter is done **atomicAdd()**.

Now, why does **histogram64** tabulate image intensities at only 6-bit granularity? It's simply a matter of resource limitations. Podlozhnyuk notes that NVIDIA says that for best efficiency, there should be between 128 and 256 threads per block. He takes the middle ground, 192. With 16K of shared memory per block, 16K/192 works out to about 85 bytes per thread. That eliminates computing a histogram for the full 8-bit image data, with 256 intensity levels, which would require 256 bytes for each thread.

Accordingly, Podlozhnyuk offers **histogram256**, which refines the process, by having one subhistogram per warp, instead of per thread. This allows the full 8-bit data, 256 levels, to be tabulated, one word devoted to each count, rather than just one byte. A subhistogram is now a table, 256 rows by 32 columns (one column for each thread in the warp), with each table entry being 4 bytes (1 byte is not sufficient, as 32 threads are tabulating with it).

15.3 Clustering

Suppose you have data consisting of (X,Y) pairs, which when plotted look like this:



It looks like there may be two or three groups here. What clustering algorithms do is to form groups, both their number and their membership, i.e. which data points belong to which groups.

(Note carefully that *there is no "correct" answer here. This is merely an exploratory data analysis tool.*)

Clustering is used is many diverse fields. For instance, it is used in image processing for segmentation and edge detection.

Here we have to two variables, say people's heights and weights. In general we have many variables, say p of them, so whatever clustering we find will be in p-dimensional space. No, we can't picture it very easily of p is larger than (or even equal to) 3, but we can at least identify membership, i.e. John and Mary are in group 1, Jenny is in group 2, etc. We may derive some insight from this.

There are many, many types of clustering algorithms. Here we will discuss the famous **k-means** algorithm, developed by Prof. Jim MacQueen of the UCLA business school.

The method couldn't be simpler. Choose k, the number of groups you want to form, and then run this:

```
# form initial groups from the first k data points (or choose randomly)
for i = 1,...,k:
   group[i] = (x[i],y[i])
   center[i] = (x[i],y[i])
do:
   for j = 1,...,n:
      find the closest center[i] to (x[j],y[j])
      cl[j] = the i you got in the previous line
   for i = 1,...,k:
      group[i] = all (x[j],y[j]) such that cl[j] = i
      center[i] = average of all (x,y) in group[i]
until group memberships do not change from one iteration to the next
```

Definitions of terms:

- *Closest* means in p-dimensional space, with the usual Euclidean distance: The distance from $(a_1, ..., a_p$ to $(b_1, ..., b_p$ is

$$\sqrt{(b_1 - a_1)^2 + ... + (b_p - a_p)^2} \tag{15.6}$$

  Other distance definitions could be used too, of course.

- The *center* of a group is its **centroid**, which is a fancy name for taking the average value in each component of the data points in the group. If p = 2, for example, the center consists of the point whose X coordinate is the average X value among members of the group, and whose Y coordinate is the average Y value in the group.

15.3.1 Example: k-Means Clustering in R

In terms of parallelization, again we have an embarrassingly parallel problem. Here's **snow** code for it:

```
# snow version of k-means clustering problem

# returns distances from x to each vector in y;
# here x is a single vector and y is a bunch of them
```

```
#
# define distance between 2 points to be the sum of the absolute values
# of their componentwise differences; e.g. distance between (5,4.2) and
# (3,5.6) is 2 + 1.4 = 3.4
dst <- function(x,y) {
   tmpmat <- matrix(abs(x-y),byrow=T,ncol=length(x)) # note recycling
   rowSums(tmpmat)
}

# will check this worker's mchunk matrix against currctrs, the current
# centers of the groups, returning a matrix; row j of the matrix will
# consist of the vector sum of the points in mchunk closest to j-th
# current center, and the count of such points
findnewgrps <- function(currctrs) {
   ngrps <- nrow(currctrs)
   spacedim <- ncol(currctrs) # what dimension space are we in?
   # set up the return matrix
   sumcounts <- matrix(rep(0,ngrps*(spacedim+1)),nrow=ngrps)
   for (i in 1:nrow(mchunk)) {
      dsts <- dst(mchunk[i,],t(currctrs))
      j <- which.min(dsts)
      sumcounts[j,] <- sumcounts[j,] + c(mchunk[i,],1)
   }
   sumcounts
}

parkm <- function(cls,m,niters,initcenters) {
   n <- nrow(m)
   spacedim <- ncol(m) # what dimension space are we in?
   # determine which worker gets which chunk of rows of m
   options(warn=-1)
   ichunks <- split(1:n,1:length(cls))
   options(warn=0)
   # form row chunks
   mchunks <- lapply(ichunks,function(ichunk) m[ichunk,])
   mcf <- function(mchunk) mchunk <<- mchunk
   # send row chunks to workers; each chunk will be a global variable at
   # the worker, named mchunk
   invisible(clusterApply(cls,mchunks,mcf))
   # send dst() to workers
   clusterExport(cls,"dst")
   # start iterations
   centers <- initcenters
```

```
48    for (i in 1:niters) {
49       sumcounts <- clusterCall(cls,findnewgrps,centers)
50       tmp <- Reduce("+",sumcounts)
51       centers <- tmp[,1:spacedim] / tmp[,spacedim+1]
52       # if a group is empty, let's set its center to 0s
53       centers[is.nan(centers)] <- 0
54    }
55    centers
56 }
```

15.4 Principal Component Analysis (PCA)

Consider data consisting of (X,Y) pairs as we saw in Section 15.3. Suppose X and Y are highly correlated with each other. Then for some constants c and d,

$$Y \approx c + dX \tag{15.7}$$

Then in a sense there is really just one random variable here, as the second is nearly equal to some linear combination of the first. The second provides us with almost no new information, once we have the first. In other words, even though the vector (X,Y) roams in *two*-dimensional space, it usually sticks close to a *one*-dimensional object, namely the line (15.7).

Now think again of p variables. It may be the case that there exist r < p variables, consisting of linear combinations of the p variables, that carry most of the information of the full set of p variables. If r is much less than p, we would prefer to work with those r variables. In data mining, this is called **dimension reduction**.

It can be shown that we can find these r variables by finding the r eigenvectors corresponding to the r largest eigenvalues of a certain matrix. So again we have a matrix formulation, and thus parallelizing the problem can be done easily by using methods for parallel matrix operations. We discussed parallel eigenvector algorithms in Section 12.6.

15.5 Monte Carlo Simulation

Monte Carlo simulation is typically (though not always) used to find probabilistic quantities such as probabilities and expected values. Consider a simple example problem:

> An urn contains blue, yellow and green marbles, in numbers 5, 12 and 13, respectively. We choose 6 marbles at random. What is the probability that we get more yellow marbles than than green and more green than blue?

We could find the approximate answer by simulation:

```
count = 0
for i = 1,...,n
   simulate drawing 6 marbles
```

```
  if yellows > greens > blues then count = count + 1
calculate approximate probability as count/n
```

The larger n is, the more accurate will be our approximate probability.

At first glance, this problem seems quite embarrassingly parallel. Say we are on a shared memory machine running 10 threads and wish to have n = 100000. Then we simply have each of our threads run the above code with n = 10000, and then average our 10 results.

The trouble with this, though, is that it assumes that the random numbers used by each thread are independent of the others. A naive approach, say by calling **random()** in the C library, will not achieve such independence. With some random number libraries, in fact, you'll get the same stream for each thread, certainly not what you want.

A number of techniques have been developed for generating parallel independent random number streams. We will not pursue the technical details here, but will give links to code for them.

- The NVIDIA CUDA SDK includes a parallel random number generator, the Mersenne Twister. The CURAND library has more.
- RngStream can be used with, for example, OpenMP and MPI.
- SPRNG is aimed at MPI, but apparently usable in shared memory settings as well. Rsprng is an R interface to SPRNG.
- OpenMP: An OpenMP version of the Mersenne Twister is available at `http://www.pgroup.com/lit/articles/insider/v2n2a4.htm`. Other parallel random number generators for OpenMP are available via a Web search.

There are many, many more.

第 16 章　Parallel Python Threads and Multiprocessing Modules

(Francis Hsu contributed sections of this chapter.)

There are a number of ways to write parallel Python code.[①]

16.1 The Python Threads and Multiprocessing Modules

Python's thread system builds on the underlying OS threads. They are thus pre-emptible. Note, though, that Python adds its own threads manager on top of the OS thread system; see Section 16.1.3.

16.1.1 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.

The `thread` Module

The example here involves a client/server pair.[②] 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**:

```
1  # simple illustration of thread module
2
3  # two clients connect to server; each client repeatedly sends a letter,
4  # stored in the variable k, which the server appends to a global string
5  # v, and reports v to the client; k = '' means the client is dropping
6  # out; when all clients are gone, server prints the final string v
7
8  # this is the client; usage is
9
```

[①] This chapter is shared by two of my open source books: `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf` and `http://heather.cs.ucdavis.edu/~matloff/Python/PLN/FastLanePython.pdf`. If you wish to more about the topics covered in the book other than the one you are now reading, please check the other!

[②] It is preferable here that the reader be familiar with basic network programming. See my tutorial at `http://heather.cs.ucdavis.edu/~matloff/Python/PLN/FastLanePython.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.

```
#    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()
```

```
34      nclntlock.acquire()
35      nclnt -= 1
36      nclntlock.release()
37
38  # set up Internet TCP socket
39  lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40
41  port = int(sys.argv[1])  # server port number
42  # bind lstn socket to this port
43  lstn.bind(('', port))
44  # start listening for contacts from clients (at most 2 at a time)
45  lstn.listen(5)
46
47  # initialize concatenated string, v
48  v = ''
49  # set up a lock to guard v
50  vlock = thread.allocate_lock()
51
52  # nclnt will be the number of clients still connected
53  nclnt = 2
54  # set up a lock to guard nclnt
55  nclntlock = thread.allocate_lock()
56
57  # accept calls from the clients
58  for i in range(nclnt):
59      # wait for call, then get a new socket to use for this client,
60      #    and get the client's address/port tuple (though not used)
61      (clnt,ap) = lstn.accept()
62      # start thread for this client, with serveclient() as the thread's
63      #    function, with parameter clnt; note that parameter set must be
64      #    a tuple; in this case, the tuple is of length 1, so a comma is
65      #    needed
66      thread.start_new_thread(serveclient,(clnt,))
67
68  # shut down the server socket, since it's not needed anymore
69  lstn.close()
70
71  # wait for both threads to finish
72  while nclnt > 0: pass
73
74  print 'the final value of v is', v
```

Make absolutely sure to run the programs before proceeding further.[③] Here is how to do this:

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**.[④] 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

[③] You can get them from the **.tex** source file for this tutorial, located wherever your picked up the **.pdf** version.

[④] 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.

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 send 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.[5].

So, let's see the technical details. We start with the "main" program.[6]

```
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,))
```

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

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

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

```
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 16.1.3, 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.[7]

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 again 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):
```

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

```
25              prime[i*k] = 0
26      print 'thread', tn, 'exiting; processed', nk, 'values of k'
27      donelock[tn].release()
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 16.1.1, we didn't want the main thread to exit until the child threads were done.[8] 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.

[8] 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.

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, to be discussed later, 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.

```
1  import sys
2  import time
3  import os
4  import thread
5
6  class glbls:
7     acctimes = []  # access times
8     acclock = thread.allocate_lock()  # lock to guard access time data
9     nextprobe = 0  # index of next site to probe
10    nextprobelock = thread.allocate_lock()  # lock to guard access time data
11    sites = open(sys.argv[1]).readlines()  # the sites to monitor
12    ww = int(sys.argv[2])  # window width
13
14 def probe(me):
15    if me > 0:
16       while 1:
17          # determine what site to probe next
18          glbls.nextprobelock.acquire()
19          i = glbls.nextprobe
20          i1 = i + 1
21          if i1 >= len(glbls.sites): i1 = 0
22          glbls.nextprobe = i1
23          glbls.nextprobelock.release()
24          # do probe
25          t1 = time.time()
26          os.system('wget --spider -q '+glbls.sites[i1])
27          t2 = time.time()
28          accesstime = t2 - t1
29          glbls.acclock.acquire()
30          # list full yet?
31          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()

```

The `threading` Module

The program below treats the same network client/server application considered in Section 16.1.1, 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 16.1.1 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 16.1.1 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'

```

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

16.1.2 Condition Variables

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.

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**.

16.1.3 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.

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.

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.

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**.

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.

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.

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

[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.

[10] This is the "machine language" for the Python virtual machine.

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.[1]

The GIL

In the case of CPython (but not Jython or Iron Python), 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.

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

[1] The author thanks Alex Martelli for a helpful clarification.

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.

Most importantly, the presence of the GIL means that two Python threads (spawned from the same program) cannot run at the same time—*even on a multicore machine*. This has been the subject of great controversy.

Implications for Randomness and Need for Locks

I mentioned in Section 16.1.3 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.

16.1.4 The `multiprocessing` Module

CPython's GIL is the subject of much controversy. As we saw in Section 16.1.3, 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), and for which I did in R as a package **Rdsm** some time later. 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:

```
1 # dice probability finder, based on Python multiprocessing class
2
3 # usage:  python DiceProb.py n k nreps nthreads
4 #   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.

16.1.5 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**:

```
1  #!/usr/bin/env python
2
3  # prime number counter, based on Python multiprocessing class with
4  # Queue
5
6  # usage:  python PrimeThreading.py n nthreads
7  #   where we wish the count of the number of primes from 2 to n, and to
8  #   use nthreads to do the work
9
10 # uses Sieve of Erathosthenes:  write out all numbers from 2 to n, then
11 # cross out all the multiples of 2, then of 3, then of 5, etc., up to
12 # sqrt(n); what's left at the end are the primes
13
14 import sys
15 import math
16 from multiprocessing import Process, Array, Queue
17
18 class glbls:  # globals, other than shared
19    n = int(sys.argv[1])
20    nthreads = int(sys.argv[2])
21    thrdlist = []  # list of all instances of this class
22
23 def prmfinder(id,prm,nxtk):
24    nk = 0  # count of k's done by this thread, to assess load balance
25    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)
      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
```

```
14
15 class glbls:  # globals, other than shared
16    nthreads = int(sys.argv[2])
17    thrdlist = []  # list of all instances of this class
18    r = random.Random(9876543)
19
20 def sortworker(id,x,q):
21    chunkinfo = q.get()
22    i = chunkinfo[0]
23    j = chunkinfo[1]
24    k = chunkinfo[2]
25    if k < glbls.nthreads - 1:  # need more splitting?
26       splitpt = separate(x,i,j)
27       q.put((splitpt+1,j,k+1))
28       # now, what do I sort?
29       rightend = splitpt + 1
30    else: rightend = j
31    tmp = x[i:(rightend+1)]  # need copy, as Array type has no sort() method
32    tmp.sort()
33    x[i:(rightend+1)] = tmp
34
35 def separate(xc, low, high):  # common algorithm; see Wikipedia
36    pivot = xc[low]  # would be better to take, e.g., median of 1st 3 elts
37    (xc[low],xc[high]) = (xc[high],xc[low])
38    last = low
39    for i in range(low,high):
40       if xc[i] <= pivot:
41          (xc[last],xc[i]) = (xc[i],xc[last])
42          last += 1
43    (xc[last],xc[high]) = (xc[high],xc[last])
44    return last
45
46 def main():
47    tmp = []
48    n = int(sys.argv[1])
49    for i in range(n): tmp.append(glbls.r.uniform(0,1))
50    x = Array('d',tmp)
51    # work items have form (i,j,k), meaning that the given array chunk
52    # corresponds to indices i through j of x, and that this is the kth
53    # chunk that has been created, x being the 0th
54    q = Queue()  # work queue
55    q.put((0,n-1,0))
56    for i in range(glbls.nthreads):
57       p = Process(target=sortworker, args=(i,x,q))
58       glbls.thrdlist.append(p)
59       p.start()
60    for thrd in glbls.thrdlist: thrd.join()
61    if n < 25: print x[:]
62
63 if __name__ == '__main__':
64     main()
```

16.1.6 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**.

16.2 Using Python with MPI

(**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.[12] 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.

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.

```
1  # a type of quicksort; break array x (actually a Python "list") into
2  # p quicksort-style piles, based # on comparison with the first p-1
3  # elements of x, where p is the number # of MPI nodes; the nodes sort
4  # their piles, then return them to node 0, # which strings them all
5  # together into the final sorted array
6
7  import mpi  # load pyMPI module
8
9  # makes npls quicksort-style piles
10 def makepiles(x,npls):
11    pivot = x[:npls]  # we'll use the first npls elements of x as pivots,
12                      # i.e. we'll compare all other elements of x to these
13    pivot.sort()  # sort() is a member function of the Python list class
14    pls = []  # initialize piles list to empty
15    lp = len(pivot)  # length of the pivot array
16    # pls will be a list of lists, with the i-th list in pls storing the
17    # i-th pile; the i-th pile will start with ID i (to enable
18    # identification later on) and pivot[i]
19    for i in range(lp):  # i = 0,1,...lp-1
20       pls.append([i,pivot[i]])  # build up array via append() member function
21    pls.append([lp])
22    for xi in x[npls:]:  # now place each element in the rest of x into
23                         # its proper pile
24       for j in range(lp):  # j = 0,1,...,lp-1
25          if xi <= pivot[j]:
26             pls[j].append(xi)
27             break
28          elif j == lp-1: pls[lp].append(xi)
29    return pls
30
31 def main():
```

[12] If you are not familiar with Python, I have a quick tutorial at `http://heather.cs.ucdavis.edu/~matloff/python.html`.

```
   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
```

16.2.1 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 16.1.1, 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.

16.2.2 RPDB2 and Winpdb

The Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[13] 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
```

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

附录 A Review of Matrix Algebra

This book assumes the reader has had a course in linear algebra (or has self-studied it, always the better approach). This appendix is intended as a review of basic matrix algebra, or a quick treatment for those lacking this background.

A.1 Terminology and Notation

A **matrix** is a rectangular array of numbers. A **vector** is a matrix with only one row (a **row vector** or only one column (a **column vector**).

The expression, "the (i,j) element of a matrix," will mean its element in row i, column j.

Please note the following conventions:

- Capital letters, e.g. A and X, will be used to denote matrices and vectors.
- Lower-case letters with subscripts, e.g. $a_{2,15}$ and x_8, will be used to denote their elements.
- Capital letters with subscripts, e.g. A_{13}, will be used to denote submatrices and subvectors.

If A is a **square** matrix, i.e. one with equal numbers n of rows and columns, then its **diagonal** elements are a_{ii}, i = 1,...,n.

A square matrix is called **upper-triangular** if $a_{ij} = 0$ whenever $i > j$, with a corresponding definition for **lower-triangular** matrices.

The **norm** (or **length**) of an n-element vector **X** is

$$\| X \| = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{A.1}$$

A.1.1 Matrix Addition and Multiplication

- For two matrices have the same numbers of rows and same numbers of columns, addition is defined elementwise, e.g.

$$\begin{pmatrix} 1 & 5 \\ 0 & 3 \\ 4 & 8 \end{pmatrix} + \begin{pmatrix} 6 & 2 \\ 0 & 1 \\ 4 & 0 \end{pmatrix} = \begin{pmatrix} 7 & 7 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \tag{A.2}$$

- Multiplication of a matrix by a **scalar**, i.e. a number, is also defined elementwise, e.g.

$$0.4\begin{pmatrix} 7 & 7 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} = \begin{pmatrix} 2.8 & 2.8 \\ 0 & 1.6 \\ 3.2 & 3.2 \end{pmatrix} \tag{A.3}$$

- The **inner product** or **dot product** of equal-length vectors X and Y is defined to be

$$\sum_{k=1}^{n} x_k y_k \tag{A.4}$$

- The product of matrices A and B is defined if the number of rows of B equals the number of columns of A (A and B are said to be **conformable**). In that case, the (i,j) element of the product C is defined to be

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{A.5}$$

  For instance,

$$\begin{pmatrix} 7 & 6 \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \begin{pmatrix} 1 & 6 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 19 & 66 \\ 8 & 16 \\ 24 & 80 \end{pmatrix} \tag{A.6}$$

  It is helpful to visualize c_{ij} as the inner product of row i of A and column j of B, e.g. as shown in bold face here:

$$\begin{pmatrix} \mathbf{7} & \mathbf{6} \\ 0 & 4 \\ 8 & 8 \end{pmatrix} \begin{pmatrix} \mathbf{1} & 6 \\ \mathbf{2} & 4 \end{pmatrix} = \begin{pmatrix} \mathbf{7} & 70 \\ 8 & 16 \\ 8 & 80 \end{pmatrix} \tag{A.7}$$

- Matrix multiplicatin is associative and distributive, but in general not commutative:

$$A(BC) = (AB)C \tag{A.8}$$

$$A(B + C) = AB + AC \tag{A.9}$$

$$AB \neq BA \tag{A.10}$$

A.2 Matrix Transpose

- The transpose of a matrix A, denoted A' or A^T, is obtained by exchanging the rows and columns of A, e.g.

$$\begin{pmatrix} 7 & 70 \\ 8 & 16 \\ 8 & 80 \end{pmatrix}' = \begin{pmatrix} 7 & 8 & 8 \\ 70 & 16 & 80 \end{pmatrix} \tag{A.11}$$

- If A + B is defined, then

$$(A+B)' = A' + B' \tag{A.12}$$

- If A and B are conformable, then

$$(AB)' = B'A' \tag{A.13}$$

A.3 Linear Independence

Equal-length vectors X_1,...,X_k are said to be **linearly independent** if it is impossible for

$$a_1X_1 + ... + a_kX_k = 0 \tag{A.14}$$

unless all the a_i are 0.

A.4 Determinants

Let A be an nxn matrix. The definition of the determinant of A, det(A), involves an abstract formula featuring permutations. It will be omitted here, in favor of the following computational method.

Let $A_{-(i,j)}$ denote the submatrix of A obtained by deleting its ith row and jth column. Then the determinant can be computed recursively across the kth row of A as

$$det(A) = \sum_{m=1}^{n} (-1)^{k+m} det(A_{-(k,m)}) \tag{A.15}$$

where

$$det \begin{pmatrix} s & t \\ u & v \end{pmatrix} = sv - tu \tag{A.16}$$

Generally, determinants are mainly of theoretical importance, but they often can clarify one's understanding of concepts.

A.5 Matrix Inverse

- The **identity** matrix I of size n has 1s in all of its diagonal elements but 0s in all off-diagonal elements. It has the property that AI = A and IA = A whenever those products are defined.
- The A is a square matrix and AB = I, then B is said to be the **inverse** of A, denoted A^{-1}. Then BA = I will hold as well.
- A^{-1} exists if and only if its rows (or columns) are linearly independent.
- A^{-1} exists if and only if $det(A) \neq 0$.
- If A and B are square, conformable and invertible, then AB is also invertible, and

$$(AB)^{-1} = B^{-1}A^{-1} \tag{A.17}$$

A matrix U is said to be **orthogonal** if its rows each have norm 1 and are orthogonal to each other, i.e. their inner product is 0. U thus has the property that $UU' = I$ i.e. $U^{-1} = U$.

The inverse of a triangular matrix is easily obtain by something called **back substitution**.

Typically one does not compute matrix inverses directly. A common alternative is the **QR decomposition**: For a matrix A, matrices Q and R are calculated so that A = QR, where Q is an orthogonal matrix and R is upper-triangular.

If A is square and inveritble, A^{-1} is easily found:

$$A^{-1} = (QR)^{-1} = R^{-1}Q' \tag{A.18}$$

Again, though, in some cases A is part of a more complex system, and the inverse is not explicitly computed.

A.6 Eigenvalues and Eigenvectors

Let A be a square matrix.[①]

- A scalar λ and a nonzero vector X that satisfy

$$AX = \lambda X \tag{A.19}$$

  are called an **eigenvalue** and **eigenvector** of A, respectively.
- If A is symmetric and real, then it is **diagonalizable**, i.e there exists an orthogonal matrix U such that

$$U'AU = D \tag{A.20}$$

  for a diagonal matrix D. The elements of D are the eigenvalues of A, and the columns of U are the eigenvectors of A.

A.7 Matrix Algebra in R

The R programming language has extensive facilities for matrix algebra, introduced here.

Note first that R matrix subscripts, like those of vectors, begin at 1, rather than 0 as in C/C++. For instance:

```
> m <- rbind(3:4,c(1,8))
> m
     [,1] [,2]
[1,] 3 4
[2,] 1 8
> m[2,2]
[1] 8
```

[①] For nonsquare matrices, the discussion here would generalize to the topic of **singular value decomposition**.

Next, it is important to know that R uses column-major order, i.e. its elements are stored in memory column-by-column. In the case of the matrix **m** above, for instance, the element 1 will be the second one in the internal memory storage of **m**, while the 8 will be the fourth.

This is also reflected in how R "inputs" data when a matrix is constructed, e.g.

```
> d <- matrix(c(1,-1,0,0,3,8),nrow=2)
> d
     [,1] [,2] [,3]
[1,] 1 0 3
[2,] -1 0 8
```

The R matrix type is a special case of vectors:

```
> d[5] # 5th element, i.e. row 1, column 3
[1] 3
```

A linear algebra vector can be formed as an R vector, or as a one-row or one-column matrix. If you use it in a matrix product, R will usually be able to figure out whether you mean it to be a row or a column.

```
> # constructing matrices
> a <- rbind(1:3,10:12)
> a
     [,1] [,2] [,3]
[1,] 1 2 3
[2,] 10 11 12
> b <- matrix(1:9,ncol=3)
> b
     [,1] [,2] [,3]
[1,] 1 4 7
[2,] 2 5 8
[3,] 3 6 9
# multiplication, addition etc.
> c <- a %*% b
> c
     [,1] [,2] [,3]
[1,] 14 32 50
[2,] 68 167 266
> c + matrix(c(1,-1,0,0,3,8),nrow=2) # 2 different c's!
     [,1] [,2] [,3]
[1,] 15 32 53
[2,] 67 167 274
> c %*% c(1,5,6)
     [,1]
[1,] 474
```

```
[2,] 2499
> t(a) # matrix transpose
     [,1] [,2]
[1,] 1 10
[2,] 2 11
[3,] 3 12
> # matrix inverse
> u <- matrix(runif(9),nrow=3)
> u
          [,1] [,2] [,3]
[1,] 0.08446154 0.86335270 0.6962092
[2,] 0.31174324 0.35352138 0.7310355
[3,] 0.56182226 0.02375487 0.2950227
> uinv <- solve(u)
> uinv
          [,1] [,2] [,3]
[1,] 0.5818482 -1.594123 2.576995
[2,] 2.1333965 -2.451237 1.039415
[3,] -1.2798127 3.233115 -1.601586
> u %*% uinv # check, but note roundoff error
            [,1] [,2] [,3]
[1,] 1.000000e+00 -1.680513e-16 -2.283330e-16
[2,] 6.651580e-17 1.000000e+00 4.412703e-17
[3,] 2.287667e-17 -3.539920e-17 1.000000e+00
> # eigenvalues and eigenvectors
> eigen(u)
$values
[1] 1.2456220+0.0000000i -0.2563082+0.2329172i -0.2563082-0.2329172i

$vectors
             [,1] [,2] [,3]
[1,] -0.6901599+0i -0.6537478+0.0000000i -0.6537478+0.0000000i
[2,] -0.5874584+0i -0.1989163-0.3827132i -0.1989163+0.3827132i
[3,] -0.4225778+0i 0.5666579+0.2558820i 0.5666579-0.2558820i
> # diagonal matrices (off-diagonals 0)
> diag(3)
     [,1] [,2] [,3]
[1,] 1 0 0
[2,] 0 1 0
[3,] 0 0 1
> diag((c(5,12,13)))
     [,1] [,2] [,3]
[1,] 5 0 0
```

```
[2,] 0 12 0
[3,] 0 0 13
> m
     [,1] [,2] [,3]
[1,] 5 6 7
[2,] 10 11 12
> diag(m) <- c(8,88)
> m
     [,1] [,2] [,3]
[1,] 8 6 7
[2,] 10 88 12
```

附录 B　R 语言快速入门

这里我们提供了一个 R 语言的快速入门，进一步的学习资料可以在 `http://heather.cs.ucdavis.edu/~/matloff/r.html`找到。

R 的语法和 C 很类似。R 既是面向对象的（从封装、多态和所有一切都是对象的角度来说），R 又是一门函数式语言（比如几乎没有副作用[①]，每个操作都是一个函数调用等）。

B.1　对应关系

aspect	C/C++	R
assignment	=	<- (or =)
array terminology	array	vector, matrix, array
subscripts	start at 0	start at 1
array notation	m[2][3]	m[2,3]
2-D array storage	row-major order	column-major order
mixed container	struct, members accessed by .	list, members acessed by $ or [[ ]]
return mechanism	return	return() or last value computed
primitive types	int, float, double, char, bool	integer, float, double, character, logical
logical values	true, false	TRUE, FALSE (abbreviated T, F)
mechanism for combining modules	include, link	library()
run method	batch	interactive, batch

B.2　Starting R

To invoke R, just type "R" into a terminal window. On a Windows machine, you probably have an R icon to click.

If you prefer to run from an IDE, you may wish to consider ESS for Emacs, StatET for Eclipse or RStudio, all open source. ESS is the favorite among the "hard core coder" types, while the colorful, easy-to-use, RStudio is a big general crowd pleaser. If you are already an Eclipse user, StatET will be just what you need.

R is normally run in interactive mode, with > as the prompt. Among other things, that makes it easy to try little experiments to learn from; remember my slogan, "When in doubt, try it out!"

[①] 译者注：函数的副作用指当调用函数时，除了返回函数值之外，还对主调用函数产生附加的影响，例如修改全局变量（函数外的变量）或修改参数

B.3 First Sample Programming Session

Below is a commented R session, to introduce the concepts. I had a text editor open in another window, constantly changing my code, then loading it via R's **source()** command. The original contents of the file **odd.R** were:

```
oddcount <- function(x) {
  k <- 0 # assign 0 to k
  for (n in x) {
     if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
  }
  return(k)
}
```

By the way, we could have written that last statement as simply

```
  k
```

because the last computed value of an R function is returned automatically.

The R session is shown below. You may wish to type it yourself as you go along, trying little experiments of your own along the way.[②]

```
> source("odd.R") # load code from the given file
> ls() # what objects do we have?
[1] "oddcount"
> # what kind of object is oddcount (well, we already know)?
> class(oddcount)
[1] "function"
> # while in interactive mode, and not inside a function, can print
> # any object by typing its name; otherwise use print(), e.g. print(x+y)
> oddcount # a function is an object, so can print it
function(x) {
  k <- 0 # assign 0 to k
  for (n in x) {
     if (n %% 2 == 1) k <- k+1 # %% is the modulo operator
  }
  return(k)
}

> # let's test oddcount(), but look at some properties of vectors first
> y <- c(5,12,13,8,88) # c() is the concatenate function
> y
[1] 5 12 13 8 88
```

[②] The source code for this file is at `http://heather.cs.ucdavis.edu/~matloff/MiscPLN/R5MinIntro.tex`. You can download the file, and copy/paste the text from there.

```
> y[2] # R subscripts begin at 1, not 0
[1] 12
> y[2:4] # extract elements 2, 3 and 4 of y
[1] 12 13 8
> y[c(1,3:5)] # elements 1, 3, 4 and 5
[1] 5 13 8 88
> oddcount(y) # should report 2 odd numbers
[1] 2

> # change code (in the other window) to vectorize the count operation,
> # for much faster execution
> source("odd.R")
> oddcount
function(x) {
   x1 <- (x %% 2 == 1) # x1 now a vector of TRUEs and FALSEs
   x2 <- x[x1] # x2 now has the elements of x that were TRUE in x1
   return(length(x2))
}

> # try it on subset of y, elements 2 through 3
> oddcount(y[2:3])
[1] 1
> # try it on subset of y, elements 2, 4 and 5
> oddcount(y[c(2,4,5)])
[1] 0

> # further compactify the code
> source("odd.R")
> oddcount
function(x) {
   length(x[x %% 2 == 1]) # last value computed is auto returned
}
> oddcount(y) # test it
[1] 2

# and even more compactification, making use of the fact that TRUE and
# FALSE are treated as 1 and 0
> oddcount <- function(x) sum(x %% 2 == 1)
# make sure you understand the steps that that involves: x is a vector,
# and thus x %% 2 is a new vector, the result of applying the mod 2
# operation to every element of x; then x %% 2 == 1 applies the == 1
# operation to each element of that result, yielding a new vector of TRUE
# and FALSE values; sum() then adds them (as 1s and 0s)
```

```
65
66 # we can also determine which elements are odd
67 > which(y %% 2 == 1)
68 [1] 1 3
69
70 > # now have ftn return odd count AND the odd numbers themselves, using
71 > # the R list type
72 > source("odd.R")
73 > oddcount
74 function(x) {
75    x1 <- x[x %% 2 == 1]
76    return(list(odds=x1, numodds=length(x1)))
77 }
78 > # R's list type can contain any type; components delineated by $
79 > oddcount(y)
80 $odds
81 [1] 5 13
82
83 $numodds
84 [1] 2
85
86 > ocy <- oddcount(y) # save the output in ocy, which will be a list
87 > ocy
88 $odds
89 [1] 5 13
90
91 $numodds
92 [1] 2
93
94 > ocy$odds
95 [1] 5 13
96 > ocy[[1]] # can get list elements using [[ ]] instead of $
97 [1] 5 13
98 > ocy[[2]]
99 [1] 2
```

Note that the function of the R function **function()** is to produce functions! Thus assignment is used. For example, here is what **odd.R** looked like at the end of the above session:

```
1 oddcount <- function(x) {
2    x1 <- x[x %% 2 == 1]
3    return(list(odds=x1, numodds=length(x1)))
4 }
```

We created some code, and then used **function()** to create a function object, which we assigned to **oddcount**.

Note that we eventually **vectorized** our function **oddcount()**. This means taking advantage of the vector-based, functional language nature of R, exploiting R's built-in functions instead of loops. This changes the venue from interpreted R to C level, with a potentially large increase in speed. For example:

```
> x <- runif(1000000) # 1000000 random numbers from the interval (0,1)
> system.time(sum(x))
   user system elapsed
  0.008 0.000 0.006
> system.time({s <- 0; for (i in 1:1000000) s <- s + x[i]})
   user system elapsed
  2.776 0.004 2.859
```

B.4 Second Sample Programming Session

A matrix is a special case of a vector, with added class attributes, the numbers of rows and columns.

```
> # "rowbind() function combines rows of matrices; there's a cbind() too
> m1 <- rbind(1:2,c(5,8))
> m1
     [,1] [,2]
[1,] 1 2
[2,] 5 8
> rbind(m1,c(6,-1))
     [,1] [,2]
[1,] 1 2
[2,] 5 8
[3,] 6 -1

> # form matrix from 1,2,3,4,5,6, in 2 rows; R uses column-major storage
> m2 <- matrix(1:6,nrow=2)
> m2
     [,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> ncol(m2)
[1] 3
> nrow(m2)
[1] 2
> m2[2,3] # extract element in row 2, col 3
```

```
[1] 6
# get submatrix of m2, cols 2 and 3, any row
> m3 <- m2[,2:3]
> m3
     [,1] [,2]
[1,] 3 5
[2,] 4 6

> m1 * m3 # elementwise multiplication
     [,1] [,2]
[1,] 3 10
[2,] 20 48
> 2.5 * m3 # scalar multiplication (but see below)
     [,1] [,2]
[1,] 7.5 12.5
[2,] 10.0 15.0
> m1 %*% m3 # linear algebra matrix multiplication
     [,1] [,2]
[1,] 11 17
[2,] 47 73

> # matrices are special cases of vectors, so can treat them as vectors
> sum(m1)
[1] 16
> ifelse(m2 %%3 == 1,0,m2) # (see below)
     [,1] [,2] [,3]
[1,] 0 3 5
[2,] 2 0 6
```

The "scalar multiplication" above is not quite what you may think, even though the result may be. Here's why:

In R, scalars don't really exist; they are just one-element vectors. However, R usually uses **recycling**, i.e. replication, to make vector sizes match. In the example above in which we evaluated the express `2.5 * m3`, the number 2.5 was recycled to the matrix

$$\begin{pmatrix} 2.5 & 2.5 \\ 2.5 & 2.5 \end{pmatrix} \tag{B.1}$$

in order to conform with **m3** for (elementwise) multiplication.

The **ifelse()** function is another example of vectorization. Its call has the form

```
ifelse(boolean vectorexpression1, vectorexpression2, vectorexpression3)
```

All three vector expressions must be the same length, though R will lengthen some via recycling. The action will be to return a vector of the same length (and if matrices are involved, then the result also has the same shape). Each element of the result will be set to its corresponding element in

vectorexpression2 or **vectorexpression3**, depending on whether the corresponding element in **vectorexpression1** is TRUE or FALSE.

In our example above,

```
> ifelse(m2 %%3 == 1,0,m2) # (see below)
```

the expression `m2 %%3 == 1` evaluated to the boolean matrix

$$\begin{pmatrix} T & F & F \\ F & T & F \end{pmatrix} \tag{B.2}$$

(TRUE and FALSE may be abbreviated to T and F.)

The 0 was recycled to the matrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{B.3}$$

while **vectorexpression3**, **m2**, evaluated to itself.

B.5 Third Sample Programming Session

This time, we focus on vectors and matrices.

```
> m <- rbind(1:3,c(5,12,13)) # "row bind," combine rows
> m
     [,1] [,2] [,3]
[1,] 1 2 3
[2,] 5 12 13
> t(m) # transpose
     [,1] [,2]
[1,] 1 5
[2,] 2 12
[3,] 3 13
> ma <- m[,1:2]
> ma
     [,1] [,2]
[1,] 1 2
[2,] 5 12
> rep(1,2) # "repeat," make multiple copies
[1] 1 1
> ma %*% rep(1,2) # matrix multiply
     [,1]
[1,] 3
[2,] 17
> solve(ma,c(3,17)) # solve linear system
[1] 1 1
```

```
> solve(ma) # matrix inverse
     [,1] [,2]
[1,] 6.0 -1.0
[2,] -2.5 0.5
```

B.6 The R List Type

The R **list** type is, after vectors, the most important R construct. A list is like a vector, except that the components are generally of mixed types.

B.6.1 The Basics

Here is example usage:

```
> g <- list(x = 4:6, s = "abc")
> g
$x
[1] 4 5 6

$s
[1] "abc"

> g$x # can reference by component name
[1] 4 5 6
> g$s
[1] "abc"
> g[[1]] # can reference by index, but note double brackets
[1] 4 5 6
> g[[2]]
[1] "abc"
> for (i in 1:length(g)) print(g[[i]])
[1] 4 5 6
[1] "abc"
```

B.6.2 The Reduce() Function

One often needs to combine elements of a list in some way. One approach to this is to use **Reduce()**:

```
> x <- list(4:6,c(1,6,8))
> x
[[1]]
[1] 4 5 6

```

```
[[2]]
[1] 1 6 8

> sum(x)
Error in sum(x) : invalid 'type' (list) of argument
> Reduce(sum,x)
[1] 30
```

Here **Reduce()** cumulatively applied R's **sum()** to **x**. Of course, you can use it with functions you write yourself too.

Continuing the above example:

```
> Reduce(c,x)
[1] 4 5 6 1 6 8
```

B.6.3 S3 Classes

R is an object-oriented (and functional) language. It features two types of classes, S3 and S4. I'll introduce S3 here.

An S3 object is simply a list, with a class name added as an *attribute*:

```
> j <- list(name="Joe", salary=55000, union=T)
> class(j) <- "employee"
> m <- list(name="Joe", salary=55000, union=F)
> class(m) <- "employee"
```

So now we have two objects of a class we've chosen to name **"employee"**. Note the quotation marks.

We can write class *generic functions*:

```
> print.employee <- function(wrkr) {
+ cat(wrkr$name,"\n")
+ cat("salary",wrkr$salary,"\n")
+ cat("union member",wrkr$union,"\n")
+ }
> print(j)
Joe
salary 55000
union member TRUE
> j
Joe
salary 55000
union member TRUE
```

What just happened? Well, **print()** in R is a *generic* function, meaning that it is just a placeholder for a function specific to a given class. When we printed **j** above, the R interpreter

searched for a function **print.employee()**, which we had indeed created, and that is what was executed. Lacking this, R would have used the print function for R lists, as before:

```
> rm(print.employee) # remove the function, to see what happens with print
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

attr(,"class")
[1] "employee"
```

B.6.4 Handy Utilities

R functions written by others, e.g. in base R or in the CRAN repository for user-contributed code, often return values which are class objects. It is common, for instance, to have lists within lists. In many cases these objects are quite intricate, and not thoroughly documented. In order to explore the contents of an object—even one you write yourself—here are some handy utilities:

- **names()**: Returns the names of a list.
- **str()**: Shows the first few elements of each component.
- **summary()**: General function. The author of a class **x** can write a version specific to **x**, i.e. **summary.x()**, to print out the important parts; otherwise the default will print some bare-bones information.

For example:

```
> z <- list(a = runif(50), b = list(u=sample(1:100,25), v="blue sky"))
> z
$a
 [1] 0.301676229 0.679918518 0.208713522 0.510032893 0.405027042
0.412388038
 [7] 0.900498062 0.119936222 0.154996457 0.251126218 0.928304164
0.979945937
[13] 0.902377363 0.941813898 0.027964137 0.992137908 0.207571134
0.049504986
[19] 0.092011899 0.564024424 0.247162004 0.730086786 0.530251779
0.562163986
[25] 0.360718988 0.392522242 0.830468427 0.883086752 0.009853107
0.148819125
```

```
 [31] 0.381143870 0.027740959 0.173798926 0.338813042 0.371025885
0.417984331
 [37] 0.777219084 0.588650413 0.916212011 0.181104510 0.377617399
0.856198893
 [43] 0.629269146 0.921698394 0.878412398 0.771662408 0.595483477
0.940457376
 [49] 0.228829858 0.700500359

$b
$b$u
 [1] 33 67 32 76 29 3 42 54 97 41 57 87 36 92 81 31 78 12 85 73 26 44
86 40 43

$b$v
[1] "blue sky"
> names(z)
[1] "a" "b"
> str(z)
List of 2
 $ a: num [1:50] 0.302 0.68 0.209 0.51 0.405 ...
 $ b:List of 2
  ..$ u: int [1:25] 33 67 32 76 29 3 42 54 97 41 ...
  ..$ v: chr "blue sky"
> names(z$b)
[1] "u" "v"
> summary(z)
  Length Class Mode
a 50 -none- numeric
b 2 -none- list
```

B.7 Data Frames

Another workhorse in R is the *data frame*. A data frame works in many ways like a matrix, but differs from a matrix in that it can mix data of different modes. One column may consist of integers, while another can consist of character strings and so on. Within a column, though, all elements must be of the same mode, and all columns must have the same length.

We might have a 4-column data frame on people, for instance, with columns for height, weight, age and name—3 numeric columns and 1 character string column.

Technically, a data frame is an R list, with one list element per column; each column is a vector. Thus columns can be referred to by name, using the **$** symbol as with all lists, or by column number, as with matrices. The matrix **a[i,j]** notation for the element of **a** in row **i**, column **j**, applies to data

frames. So do the **rbind()** and **cbind()** functions, and various other matrix operations, such as filtering.

Here is an example using the dataset **airquality**, built in to R for illustration purposes. You can learn about the data through R's online help, i.e.

```
> ?airquality
```

Let's try a few operations:

```
> names(airquality)
[1] "Ozone" "Solar.R" "Wind" "Temp" "Month" "Day"
> head(airquality) # look at the first few rows
  Ozone Solar.R Wind Temp Month Day
1 41 190 7.4 67 5 1
2 36 118 8.0 72 5 2
3 12 149 12.6 74 5 3
4 18 313 11.5 62 5 4
5 NA NA 14.3 56 5 5
6 28 NA 14.9 66 5 6
> airquality[5,3] # temp on the 5th day
[1] 14.3
> airquality$Wind[3] # same
[1] 12.6
> nrow(airquality) # number of days observed
[1] 153
> ncol(airquality) # number of variables
[1] 6
> airquality$Celsius <- (5/9) * (airquality[,4] - 32) # new variable
> names(airquality)
[1] "Ozone" "Solar.R" "Wind" "Temp" "Month" "Day" "Celsius"
> ncol(airquality)
[1] 7
> airquality[1:3,]
  Ozone Solar.R Wind Temp Month Day Celsius
1 41 190 7.4 67 5 1 19.44444
2 36 118 8.0 72 5 2 22.22222
3 12 149 12.6 74 5 3 23.33333
> aqjune <- airquality[airquality$Month == 6,] # filter op
> nrow(aqjune)
[1] 30
> mean(aqjune$Temp)
[1] 79.1
> write.table(aqjune,"AQJune") # write data frame to file
> aqj <- read.table("AQJune",header=T) # read it in
```

B.8 Graphics

R excels at graphics, offering a rich set of capabilities, from beginning to advanced. In addition to the functions in base R, extensive graphics packages are available, such as **lattice** and **ggplot2**.

One point of confusion for beginniners involves saving an R graph that is currently displayed on the screen to a file. Here is a function for this, which I include in my R startup file, **.Rprofile**, in my home directory:

```
pr2file
function (filename)
{
    origdev <- dev.cur()
    parts <- strsplit(filename, ".", fixed = TRUE)
    nparts <- length(parts[[1]])
    suff <- parts[[1]][nparts]
    if (suff == "pdf") {
        pdf(filename)
    }
    else if (suff == "png") {
        png(filename)
    }
    else jpeg(filename)
    devnum <- dev.cur()
    dev.set(origdev)
    dev.copy(which = devnum)
    dev.set(devnum)
    dev.off()
    dev.set(origdev)
}
```

The code, which I won't go into here, mostly involves manipulation of various R graphics devices. I've set it up so that you can save to a file of type either PDF, PNG or JPEG, implied by the file name you give.

B.9 Other Sources for Learning R

There are tons of resources for R on the Web. You may wish to start with the links at `http://heather.cs.ucdavis.edu/~matloff/r.html`.

B.10 Online Help

R's **help()** function, which can be invoked also with a question mark, gives short descriptions of the R functions. For example, typing

```
> ?rep
```

will give you a description of R's **rep()** function.

An especially nice feature of R is its **example()** function, which gives nice examples of whatever function you wish to query. For instance, typing

```
> example(wireframe())
```

will show examples—R code and resulting pictures—of **wireframe()**, one of R's 3-dimensional graphics functions.

B.11 Debugging in R

The internal debugging tool in R, **debug()**, is usable but rather primitive. Here are some alternatives:

- The RStudio IDE has a built-in debugging tool.
- The StatET IDE for R on Eclipse has a nice debugging tool. Works on all major platforms, but can be tricky to install.
- My own debugging tool, **debugR**, is extensive and easy to install, but for the time being is limited to Linux, Mac and other Unix-family systems. See *http://heather.cs.ucdavis.edu/debugR.html*.

B.12 Complex Numbers

If you have need for complex numbers, R does handle them. Here is a sample of use of the main functions of interest:

```
> za <- complex(real=2,imaginary=3.5)
> za
[1] 2+3.5i
> zb <- complex(real=1,imaginary=-5)
> zb
[1] 1-5i
> za * zb
[1] 19.5-6.5i
> Re(za)
[1] 2
> Im(za)
[1] 3.5
> za^2
[1] -8.25+14i
> abs(za)
[1] 4.031129
> exp(complex(real=0,imaginary=pi/4))
[1] 0.7071068+0.7071068i
> cos(pi/4)
[1] 0.7071068
```

```
> sin(pi/4)
[1] 0.7071068
```

Note that operations with complex-valued vectors and matrices work as usual; there are no special complex functions.

B.13 Further Reading

For further information about R as a programming language, there is my book, *The Art of R Programming: a Tour of Statistical Software Design*, NSP, 2011.

For R's statistical functions, a plethora of excellent books is available. such as *The R Book* (2nd Ed.), Michael Crowley, Wiley, 2012. I also very much like *R in a Nutshell* (2nd Ed.), Joseph Adler, O'Reilly, 2012.

附录 C Introduction to Python

NOTE: This document is the first part of my open source book on Python, `http://heather.cs.ucdavis.edu/~matloff/Python/PLN/FastLanePython.pdf`. Go there for further information.

So, let's get started with programming right away.

C.1 A 5-Minute Introductory Example

C.1.1 Example Program Code

Here is a simple, quick example. Suppose I wish to find the value of

$$g(x) = \frac{x}{1-x^2}$$

for x = 0.0, 0.1, ..., 0.9. I could find these numbers by placing the following code,

```
for i in range(10):
   x = 0.1*i
   print x
   print x/(1-x*x)
```

in a file, say **fme.py**, and then running the program by typing

```
python fme.py
```

at the command-line prompt. The output will look like this:

```
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
```

C.1.2 Python Lists

How does the program work? First, Python's **range()** function is an example of the use of **lists**, i.e. Python arrays,[①] even though not quite explicitly. Lists are absolutely fundamental to Python, so watch out in what follows for instances of the word "list"; resist the temptation to treat it as the English word "list," instead always thinking about the Python construct **list**.

Python's **range()** function returns a list of consecutive integers, in this case the list [0,1,2,3,4,5,6,7,8,9]. Note that this is official Python notation for lists—a sequence of objects (these could be all kinds of things, not necessarily numbers), separated by commas and enclosed by brackets.

C.1.3 Loops

So, the **for** statement above is equivalent to:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
```

As you can guess, this will result in 10 iterations of the loop, with **i** first being 0, then 1, etc.

The code

```
for i in [2,3,6]:
```

would give us three iterations, with **i** taking on the values 2, 3 and 6.

Python has a **while** construct too (though not an **until**).

There is also a **break** statement like that of C/C++, used to leave loops "prematurely." For example:

```
x = 5
while 1:
   x += 1
   if x == 8:
      print x
      break
```

Also very useful is the **continue** statement, which instructs the Python interpreter to skip the remainder of the current iteration of a loop. For instance, running the code

```
sum = 0
for i in [5,12,13]:
   if i < 10: continue
   sum += i
print sum
```

prints out 12+13, i.e. 25.

The **pass** statement is a "no-op," doing nothing.

C.1.4 Python Block Definition

Now focus your attention on that inoccuous-looking colon at the end of the **for** line above, which defines the start of a block. Unlike languages like C/C++ or even Perl, which use braces to

[①] I loosely speak of them as "arrays" here, but as you will see, they are more flexible than arrays in C/C++.

On the other hand, true arrays can be accessed more quickly. In C/C++, the i^{th} element of an array **X** is i words past the beginning of the array, so we can go right to it. This is not possible with Python lists, so the latter are slower to access. The NumPy add-on package for Python offers true arrays.

define blocks, Python uses a combination of a colon and indenting to define a block. I am using the colon to say to the Python interpreter,

> Hi, Python interpreter, how are you? I just wanted to let you know, by inserting this colon, that a block begins on the next line. I've indented that line, and the two lines following it, further right than the current line, in order to tell you those three lines form a block.

I chose 3-space indenting, but the amount wouldn't matter as long as I am consistent. If for example I were to write[2]

```
for i in range(10):
   print 0.1*i
      print g(0.1*i)
```

the Python interpreter would give me an error message, telling me that I have a syntax error.[3] I am only allowed to indent further-right within a given block if I have a sub-block within that block, e.g.

```
for i in range(10):
   if i%2 == 1:
      print 0.1*i
      print g(0.1*i)
```

Here I am printing out only the cases in which the variable **i** is an odd number; % is the "mod" operator as in C/C++.

Again, note the colon at the end of the **if** line, and the fact that the two **print** lines are indented further right than the **if** line.

Note also that, again unlike C/C++/Perl, there are no semicolons at the end of Python source code statements. A new line means a new statement. If you need a very long line, you can use the backslash character for continuation, e.g.

```
x = y + \
      z
```

Most of the usual C operators are in Python, including the relational ones such as the == seen here. The **0x** notation for hex is there, as is the FORTRAN **** for exponentiation.

Also, the **if** construct can be paired with **else** as usual, and you can abbreviate **else if** as **elif**.

```
>> def f(x):
...    if x > 0: return 1
...    else: return 0
...
>>> f(2)
1
>>> f(-1)
0
```

The boolean operators are **and**, **or** and **not**.

You'll see examples as we move along.

By the way, watch out for Python statements like **print a or b or c**, in which the first true (i.e. nonzero) expression is printed and the others ignored; this is a common Python idiom.

[2] Here **g()** is a function I defined earlier, not shown.

[3] Keep this in mind. New Python users are often baffled by a syntax error arising in this situation.

C.1.5 Python Also Offers an Interactive Mode

A really nice feature of Python is its ability to run in interactive mode. You usually won't do this, but it's a great way to do a quick tryout of some feature, to really see how it works. Whenever you're not sure whether something works, your motto should be, "When in doubt, try it out!", and interactive mode makes this quick and easy.

We'll also be doing a lot of that in this tutorial, with interactive mode being an easy way to do a quick illustration of a feature.

Instead of executing this program from the command line in **batch** mode as we did above, we could enter and run the code in **interactive** mode:

```
% python
>>> for i in range(10):
...     x = 0.1*i
...     print x
...     print x/(1-x*x)
...
0.0
0.0
0.1
0.10101010101
0.2
0.208333333333
0.3
0.32967032967
0.4
0.47619047619
0.5
0.666666666667
0.6
0.9375
0.7
1.37254901961
0.8
2.22222222222
0.9
4.73684210526
>>>
```

Here I started Python, and it gave me its >>> interactive prompt. Then I just started typing in the code, line by line. Whenever I was inside a block, it gave me a special prompt, "...", for that purpose. When I typed a blank line at the end of my code, the Python interpreter realized I was done, and ran the code.④

While in interactive mode, one can go up and down the command history by using the arrow keys, thus saving typing.

To exit interactive Python, hit ctrl-d.

Automatic printing: By the way, in interactive mode, just referencing or producing an object, or even an expression, without assigning it, will cause its value to print out, even without a **print** statement. For example:

④Interactive mode allows us to execute only single Python statements or evaluate single Python expressions. In our case here, we typed in and executed a single **for** statement. Interactive mode is not designed for us to type in an entire program. Technically we could work around this by beginning with something like "if 1:", making our program one large **if** statement, but of course it would not be convenient to type in a long program anyway.

```
>>> for i in range(4):
...     3*i
...
0
3
6
9
```

Again, this is true for general objects, not just expressions, e.g.:

```
>>> open('x')
<open file 'x', mode 'r' at 0xb7eaf3c8>
```

Here we opened the file **x**, which produces a file object. Since we did not assign to a variable, say **f**, for reference later in the code, i.e. we did not do the more typical

```
f = open('x')
```

the object was printed out. We'd get that same information this way:

```
>>> f = open('x')
>>> f
<open file 'x', mode 'r' at 0xb7f2a3c8>
```

C.1.6 Python As a Calculator

Among other things, this means you can use Python as a quick calculator (which I do a lot). If for example I needed to know what 5% above $88.88 is, I could type

```
% python
>>> 1.05*88.88
93.323999999999998
```

Among other things, one can do quick conversions between decimal and hex:

```
>>> 0x12
18
>>> hex(18)
'0x12'
```

If I need math functions, I must **import** the Python math library first. This is analogous to what we do in C/C++, where we must have a **#include** line for the library in our source code and must link in the machine code for the library.

We must refer to imported functions in the context of the library, in this case the math library. For example, the functions **sqrt()** and **sin()** must be prefixed by **math**:[5]

```
>>> import math
>>> math.sqrt(88)
9.3808315196468595
>>> math.sin(2.5)
0.59847214410395655
```

[5] A method for avoiding the prefix is shown in Sec. **??**.

C.2 A 10-Minute Introductory Example

C.2.1 Example Program Code

This program reads a text file, specified on the command line, and prints out the number of lines and words in the file:

```
# reads in the text file whose name is specified on the command line,
# and reports the number of lines and words

import sys

def checkline():
   global l
   global wordcount
   w = l.split()
   wordcount += len(w)

wordcount = 0
f = open(sys.argv[1])
flines = f.readlines()
linecount = len(flines)
for l in flines:
   checkline()
print linecount, wordcount
```

Say for example the program is in the file **tme.py**, and we have a text file **x** with contents

```

This is an
example of a
text file.

```

(There are five lines in all, the first and last of which are blank.)

If we run this program on this file, the result is:

```
python tme.py x
5 8
```

On the surface, the layout of the code here looks like that of a C/C++ program: First an **import** statement, analogous to **#include** (with the corresponding linking at compile time) as stated above; second the definition of a function; and then the "main" program. This is basically a good way to look at it, but keep in mind that the Python interpreter will execute everything in order, starting at the top. In executing the **import** statement, for instance, that might actually result in some code being executed, if the module being imported has some free-standing code rather than just function definitions. More on this later. Execution of the **def** statement won't execute any code for now, but the act of defining the function is considered execution.

Here are some features in this program which were not in the first example:

- use of command-line arguments
- file-manipulation mechanisms
- more on lists

- function definition
- library importation
- introduction to scope

I will discuss these features in the next few sections.

C.2.2 Command-Line Arguments

First, let's explain **sys.argv**. Python includes a **module** (i.e. library) named **sys**, one of whose member variables is **argv**. The latter is a Python list, analogous to **argv** in C/C++.[6] Element 0 of the list is the script name, in this case **tme.py**, and so on, just as in C/C++. In our example here, in which we run our program on the file **x**, **sys.argv[1]** will be the string 'x' (strings in Python are generally specified with single quote marks). Since **sys** is not loaded automatically, we needed the **import** line.

Both in C/C++ and Python, those command-line arguments are of course strings. If those strings are supposed to represent numbers, we could convert them. If we had, say, an integer argument, in C/C++ we would do the conversion using **atoi()**; in Python, we'd use **int()**. For floating-point, in Python we'd use **float()**.[7]

C.2.3 Introduction to File Manipulation

The function **open()** is similar to the one in C/C++. Our line

```
1 f = open(sys.argv[1])
```

created an object of **file** class, and assigned it to **f** .

The **readlines()** function of the **file** class returns a list (keep in mind, "list" is an official Python term) consisting of the lines in the file. Each line is a string, and that string is one element of the list. Since the file here consisted of five lines, the value returned by calling **readlines()** is the five-element list

```
1 ['','This is an','example of a','text file','']
```

(Though not visible here, there is an end-of-line character in each string.)

C.2.4 Lack of Declaration

Variables are not declared in Python. A variable is created when the first assignment to it is executed. For example, in the program **tme.py** above, the variable **flines** does not exist until the statement

```
1 flines = f.readlines()
```

is executed.

By the way, a variable which has not been assigned a value yet, such as **wordcount** at first above, has the value **None**. And this can be assigned to a variable, tested for in an **if** statement, etc.

[6] There is no need for an analog of **argc**, though. Python, being an object-oriented language, treats lists as objects, The length of a list is thus incorporated into that object. So, if we need to know the number of elements in **argv**, we can get it via **len(argv)**.

[7] In C/C++, we could use **atof()** if it were available, or **sscanf()**.

C.2.5 Locals Vs. Globals

Python does not really have global variables in the sense of C/C++, in which the scope of a variable is an entire program. We will discuss this further in Section **??**, but for now assume our source code consists of just a single **.py** file; in that case, Python does have global variables pretty much like in C/C++ (though with important differences).

Python tries to infer the scope of a variable from its position in the code. If a function includes any code which assigns to a variable, then that variable is assumed to be local, unless we use the **global** keyword. So, in the code for **checkline()**, Python would assume that **l** and **wordcount** are local to **checkline()** if we had not specified **global**.

Use of global variables simplifies the presentation here, and I personally believe that the unctuous criticism of global variables is unwarranted. (See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.) In fact, in one of the major types of programming, **threads**, use of globals is basically *mandatory*.

You may wish, however, to at least group together all your globals into a class, as I do. See Appendix **??**.

C.2.6 A Couple of Built-In Functions

The function **len()** returns the number of elements in a list. In the **tme.py** example above, we used this to find the number of lines in the file, since **readlines()** returned a list in which each element consisted of one line of the file.

The method **split()** is a member of the **string** class.[8] It splits a string into a list of words, for example.[9] So, for instance, in **checkline()** when **l** is 'This is an' then the list **w** will be equal to ['This','is','an']. (In the case of the first line, which is blank, **w** will be equal to the empty list, [].)

C.3 Types of Variables/Values

As is typical in scripting languages, type in the sense of C/C++ **int** or **float** is not declared in Python. However, the Python interpreter does internally keep track of the type of all objects. Thus Python variables don't have types, but their values do. In other words, a variable **X** might be bound to (i.e. point to) an integer in one place in your program and then be rebound to a class instance at another point.

Python's types include notions of scalars, **sequences** (lists or **tuples**) and dictionaries (associative arrays, discussed in Sec. C.6), classes, function, etc.

C.4 String Versus Numerical Values

Unlike Perl, Python does distinguish between numbers and their string representations. The functions **eval()** and **str()** can be used to convert back and forth. For example:

```
1 >>> 2 + '1.5'
2 Traceback (most recent call last):
```

[8] Member functions of classes are referred to as **methods**.

[9] The default is to use blank characters as the splitting criterion, but other characters or strings can be used.

```
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> 2 + eval('1.5')
3.5
>>> str(2 + eval('1.5'))
'3.5'
```

There are also **int()** to convert from strings to integers, and **float()**, to convert from strings to floating-point values:

```
>>> n = int('32')
>>> n
32
>>> x = float('5.28')
>>> x
5.2800000000000002
```

See also Section C.5.3.

C.5 Sequences

Lists are actually special cases of **sequences**, which are all array-like but with some differences. Note though, the commonalities; all of the following (some to be explained below) apply to any sequence type:

- the use of brackets to denote individual elements (e.g. **x[i]**)
- the built-in **len()** function to give the number of elements in the sequence[10]
- **slicing** operations, i.e. the extraction of subsequences
- use of + and * operators for concatenation and replication

C.5.1 Lists (Quasi-Arrays)

As stated earlier, lists are denoted by brackets and commas. For instance, the statement

```
x = [4,5,12]
```

would set **x** to the specified 3-element array.

Lists may grow dynamically, using the **list** class' **append()** or **extend()** functions. For example, if after the abovfe statement we were to execute

```
x.append(-2)
```

x would now be equal to [4,5,12,-2].

A number of other operations are available for lists, a few of which are illustrated in the following code:

```
>>> x = [5,12,13,200]
>>> x
[5, 12, 13, 200]
>>> x.append(-2)
>>> x
```

[10] This function is applicable to dictionaries too.

```
[5, 12, 13, 200, -2]
>>> del x[2]
>>> x
[5, 12, 200, -2]
>>> z = x[1:3]  # array "slicing": elements 1 through 3-1 = 2
>>> z
[12, 200]
>>> yy = [3,4,5,12,13]
>>> yy[3:]  # all elements starting with index 3
[12, 13]
>>> yy[:3]  # all elements up to but excluding index 3
[3, 4, 5]
>>> yy[-1]  # means "1 item from the right end"
13
>>> x.insert(2,28)  # insert 28 at position 2
>>> x
[5, 12, 28, 200, -2]
>>> 28 in x  # tests for membership; 1 for true, 0 for false
1
>>> 13 in x
0
>>> x.index(28)  # finds the index within the list of the given value
2
>>> x.remove(200)  # different from "delete," since it's indexed by value
>>> x
[5, 12, 28, -2]
>>> w = x + [1,"ghi"]  # concatenation of two or more lists
>>> w
[5, 12, 28, -2, 1, 'ghi']
>>> qz = 3*[1,2,3]  # list replication
>>> qz
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> x = [1,2,3]
>>> x.extend([4,5])
>>> x
[1, 2, 3, 4, 5]
>>> y = x.pop(0)  # deletes and returns 0th element
>>> y
1
>>> x
[2, 3, 4, 5]
>>> t = [5,12,13]
>>> t.reverse()
>>> t
[13, 12, 5]
```

We also saw the **in** operator in an earlier example, used in a **for** loop.

A list could include mixed elements of different types, including other lists themselves.

The Python idiom includes a number of common "Python tricks" involving sequences, e.g. the following quick, elegant way to swap two variables **x** and **y**:

```
>>> x = 5
>>> y = 12
>>> [x,y] = [y,x]
>>> x
12
>>> y
5
```

Multidimensional lists can be implemented as lists of lists. For example:

```
>>> x = []
>>> x.append([1,2])
>>> x
[[1, 2]]
>>> x.append([3,4])
>>> x
[[1, 2], [3, 4]]
>>> x[1][1]
4
```

But be careful! Look what can go wrong:

```
>>> x = 4*[0]
>>> y = 4*[x]
>>> y
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> y[0][2]
0
>>> y[0][2] = 1
>>> y
[[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
```

The problem is that that assignment to **y** was really a list of four references to the same thing (**x**). When the object pointed to by **x** changed, then all four rows of **y** changed.

The Python Wikibook (`http://en.wikibooks.org/wiki/Python_Programming/Lists`) suggests a solution, in the form of **list comprehensions**, which we cover in Section **??**:

```
>>> z = [[0]*4 for i in range(5)]
>>> z
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> z[0][2] = 1
>>> z
[[0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

C.5.2 Tuples

Tuples are like lists, but are **immutable**, i.e. unchangeable. They are enclosed by parentheses or nothing at all, rather than brackets. The parentheses are mandatory if there is an ambiguity without them, e.g. in function arguments. A comma must be used in the case of empty or single tuple, e.g. **(,)** and **(5,)**.

The same operations can be used, except those which would change the tuple. So for example

```
x = (1,2,'abc')
print x[1]  # prints 2
print len(x)  # prints 3
x.pop()  # illegal, due to immutability
```

A nice function is **zip()**, which strings together corresponding components of several lists, producing tuples, e.g.

```
>>> zip([1,2],['a','b'],[168,168])
[(1, 'a', 168), (2, 'b', 168)]
```

C.5.3 Strings

Strings are essentially tuples of character elements. But they are quoted instead of surrounded by parentheses, and have more flexibility than tuples of character elements would have.

Strings As Turbocharged Tuples

Let's see some examples of string operations:

```
>>> x = 'abcde'
>>> x[2]
'c'
>>> x[2] = 'q'  # illegal, since strings are immmutable
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> x = x[0:2] + 'q' + x[3:5]
>>> x
'abqde'
```

(You may wonder why that last assignment

```
>>> x = x[0:2] + 'q' + x[3:5]
```

does not violate immmutability. The reason is that **x** is really a pointer, and we are simply pointing it to a new string created from old ones. See Section **??**.)

As noted, strings are more than simply tuples of characters:

```
>>> x.index('d')  # as expected
3
>>> 'd' in x  # as expected
1
>>> x.index('de')  # pleasant surprise
3
```

As can be seen, the **index()** function from the **str** class has been overloaded, making it more flexible.

There are many other handy functions in the **str** class. For example, we saw the **split()** function earlier. The opposite of this function is **join()**. One applies it to a string, with a sequence of strings as an argument. The result is the concatenation of the strings in the sequence, with the original string between each of them:[11]

```
>>> '---'.join(['abc','de','xyz'])
'abc---de---xyz'
>>> q = '\n'.join(('abc','de','xyz'))
>>> q
'abc\nde\nxyz'
>>> print q
abc
de
xyz
```

Here are some more:

[11] The example here shows the "new" usage of **join()**, now that string methods are built-in to Python. See discussion of "new" versus "old" below.

```
>>> x = 'abc'
>>> x.upper()
'ABC'
>>> 'abc'.upper()
'ABC'
>>> 'abc'.center(5) # center the string within a 5-character set
' abc '
>>> 'abc de f'.replace(' ','+')
'abc+de+f'
>>> x = 'abc123'
>>> x.find('c1')  # find index of first occurrence of 'c1' in x
2
>>> x.find('3')
5
>>> x.find('1a')
-1
```

A very rich set of functions for string manipulation is also available in the **re** ("regular expression") module.

The **str** class is built-in for newer versions of Python. With an older version, you will need a statement

```
import string
```

That latter class does still exist, and the newer **str** class does not quite duplicate it.

Formatted String Manipulation

String manipulation is useful in lots of settings, one of which is in conjunction with Python's **print** command. For example,

```
print "the factors of 15 are %d and %d" % (3,5)
```

prints out

```
the factors of 15 are 3 and 5
```

The **%d** of course is the integer format familiar from C/C++.

But actually, the above action is a string issue, not a print issue. Let's see why. In

```
print "the factors of 15 are %d and %d" % (3,5)
```

the portion

```
"the factors of 15 are %d and %d" % (3,5)
```

is a string operation, producing a new string; the **print** simply prints that new string.

For example:

```
>>> x = "%d years old" % 12
```

The variable **x** now is the string '12 years old'.

This is another very common idiom, quite powerful.[12]

Note the importance above of writing '(3,5)' rather than '3,5'. In the latter case, the % operator would think that its operand was merely 3, whereas it needs a 2-element tuple. Recall that parentheses enclosing a tuple can be omitted as long as there is no ambiguity, but that is not the case here.

[12] Some C/C++ programmers might recognize the similarity to **sprintf()** from the C library.

C.6 Dictionaries (Hashes)

Dictionaries are **associative arrays**. The technical meaning of this will be discussed below, but from a pure programming point of view, this means that one can set up arrays with non-integer indices. The statement

```
x = {'abc':12,'sailing':'away'}
```

sets **x** to what amounts to a 2-element array with **x['abc']** being 12 and **x['sailing']** equal to 'away'. We say that **'abc'** and **'sailing'** are **keys**, and 12 and 'away' are **values**. Keys can be any immmutable object, i.e. numbers, tuples or strings.[13] Use of tuples as keys is quite common in Python applications, and you should keep in mind that this valuable tool is available.

Internally, **x** here would be stored as a 4-element array, and the execution of a statement like

```
w = x['sailing']
```

would require the Python interpreter to search through that array for the key 'sailing'. A linear search would be slow, so internal storage is organized as a hash table. This is why Perl's analog of Python's dictionary concept is actually called a **hash**.

Here are examples of usage of some of the member functions of the **dictionary** class:

```
>>> x = {'abc':12,'sailing':'away'}
>>> x['abc']
12
>>> y = x.keys()
>>> y
['abc', 'sailing']
>>> z = x.values()
>>> z
[12, 'away']
x['uv'] = 2
>>> x
{'abc': 12, 'uv': 2, 'sailing': 'away'}
```

Note how we added a new element to **x** near the end.

The keys need not be tuples. For example:

```
>>> x
{'abc': 12, 'uv': 2, 'sailing': 'away'}
>>> f = open('z')
>>> x[f] = 88
>>> x
{<open file 'z', mode 'r' at 0xb7e6f338>: 88, 'abc': 12, 'uv': 2, 'sailing': 'away'}
```

Deletion of an element from a dictionary can be done via **pop()**, e.g.

```
>>> x.pop('abc')
12
>>> x
{<open file 'x', mode 'r' at 0xb7e6f338>: 88, 'uv': 2, 'sailing': 'away'}
```

The **in** operator works on dictionary keys, e.g.

[13] Now one sees a reason why Python distinguishes between tuples and lists. Allowing mutable keys would be an implementation nightmare, and probably lead to error-prone programming.

```
>>> x = {'abc': 12, 'uv': 2, 'sailing': 'away'}
>>> 'uv' in x
True
>>> 2 in x
False
```

C.7 Extended Example: Computing Final Grades

```
# computes and records final grades

# input line format:

#    name and misc. info, e.g. class level
#    Final Report grade
#    Midterm grade
#    Quiz grades
#    Homework grades

# comment lines, beginning with #, are ignored for computation but are
# printed out; thus various notes can be put in comment lines; e.g.
# notes on missed or makeup exams

# usage:

#   python FinalGrades.py input_file nq nqd nh wts

#      where there are nq Quizzes, the lowest nqd of which will be
#      deleted; nh Homework assignments; and wts is the set of weights
#      for Final Report, Midterm, Quizzes and Homework

# outputs to stdout the input file with final course grades appended;
# the latter are numerical only, allowing for personal inspection of
# "close" cases, etc.

import sys

def convertltr(lg):  # converts letter grade lg to 4-point-scale
   if lg == 'F': return 0
   base = lg[0]
   olg = ord(base)
   if len(lg) > 2 or olg < ord('A') or olg > ord('D'):
      print lg, 'is not a letter grade'
      sys.exit(1)
   grade = 4 - (olg-ord('A'))
   if len(lg) == 2:
      if lg[1] == '+': grade += 0.3
      elif lg[1] == '-': grade -= 0.3
      else:
         print lg, 'is not a letter grade'
         sys.exit(1)
   return grade

def avg(x,ndrop):
   tmp = []
   for xi in x: tmp.append(convertltr(xi))
   tmp.sort()
   tmp = tmp[ndrop:]
   return float(sum(tmp))/len(tmp)
```

```
51
52 def main():
53    infile = open(sys.argv[1])
54    nq = int(sys.argv[2])
55    nqd = int(sys.argv[3])
56    nh = int(sys.argv[4])
57    wts = []
58    for i in range(4): wts.append(float(sys.argv[5+i]))
59    for line in infile.readlines():
60       toks = line.split()
61       if toks[0] != '#':
62          lw = len(toks)
63          startpos = lw - nq - nh - 3
64          # Final Report
65          frgrade = convertltr(toks[startpos])
66          # Midterm letter grade (skip over numerical grade)
67          mtgrade = convertltr(toks[startpos+2])
68          startquizzes = startpos + 3
69          qgrade = avg(toks[startquizzes:startquizzes+nq],nqd)
70          starthomework = startquizzes + nq
71          hgrade = avg(toks[starthomework:starthomework+nh],0)
72          coursegrade = 0.0
73          coursegrade += wts[0] * frgrade
74          coursegrade += wts[1] * mtgrade
75          coursegrade += wts[2] * qgrade
76          coursegrade += wts[3] * hgrade
77          print line[:len(line)-1], coursegrade
78       else:
79          print line[:len(line)-1]
80
81 main()
```