- Hadoop/MapReduce Computing (Chapter **??**) is basically a scatter/gather operation.
- The **snow** package (Section 1.4.8.1) for the R language is also a scatter/gather operation.

1.4.8.1 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.

NOTE: 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:

```
1  > library(parallel)
2  > c2 <- makePSOCKcluster(rep("localhost",2))
3  > c2
4  socket cluster with 2 nodes on host  localhost
5  > mmul
6  function(cls,u,v) {
7     rowgrps <- splitIndices(nrow(u),length(cls))
8     grpmul <- function(grp) u[grp,] %*% v
9     mout <- clusterApply(cls,rowgrps,grpmul)
10    Reduce(c,mout)
11 }
12 > a <- matrix(sample(1:50,16,replace=T),ncol=2)
13 > a
14       [,1] [,2]
15 [1,]   34   41
16 [2,]   10   28
17 [3,]   44   23
18 [4,]    7   29
19 [5,]    6   24
20 [6,]   28   29
21 [7,]   21    1
22 [8,]   38   30
23 > b <- c(5,-2)
24 > b
25 [1]  5 -2
26 > a %*% b  # serial multiply
27       [,1]
28 [1,]   88
29 [2,]   -6
30 [3,]  174
31 [4,]  -23
32 [5,]  -18
```

```
33  [6,]   82
34  [7,]  103
35  [8,]  130
36  > clusterExport(c2,c('a','b'))  # send b to workers
37  > clusterEvalQ(c2,b)  # check that they have it
38  [[1]]
39  [1]  5 -2
40
41  [[2]]
42  [1]  5 -2
43
44  > mmul(c2,a,b)  # test our parallel code
45  [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; those R processes may be running on different machines (i.e. a real cluster), or on a multicore machine, or a combination of the two.

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)though two of them may be the batch form of R, called Rscript). An entry for a worker may look like

```
/usr/local/lib/R/bin/exec/R --slave --no-restore
   -e parallel:::.slaveRSOCK()
   --args MASTER=localhost PORT=11526 OUT=/dev/null
   TIMEOUT=2592000 METHODS=TRUE XDR=TRUE
```

So, this R process is running the **.slaveRSOCK()** function in the **parallel** package, on a TCP/IP socket at port 11526.

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:

```
1 mmul <- function(cls,u,v) {
2     rowgrps <- splitIndices(nrow(u),length(cls))
3     grpmul <- function(grp) u[grp,] %*% v
4     mout <- clusterApply(cls,rowgrps,grpmul)
5     Reduce(c,mout)
6 }
```

As discussed in Section 1.4.1, 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 (or a vector, which is promotable to a lists), 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:

```
1 mmul1 <- function(cls,u,v) {
2     rowgrps <- splitIndices(nrow(u),length(cls))
3     uchunks <- Map(function(grp) u[grp,],rowgrps)
4     mulchunk <- function(uc) uc %*% v
5     mout <- clusterApply(cls,uchunks,mulchunk)
6     Reduce(c,mout)
7 }
```

Let's test it:

```
1  > a <- matrix(sample(1:50,16,replace=T),ncol=2)
2  > b <- c(5,-2)
3  > clusterExport(c2,"b")  # don't send a
4   a
5        [,1] [,2]
6  [1,]   10   26
7  [2,]    1   34
8  [3,]   49   30
9  [4,]   39   41
10 [5,]   12   14
11 [6,]    2   30
12 [7,]   33   23
13 [8,]   44    5
14 > a %*% b
15       [,1]
16 [1,]   -2
17 [2,]  -63
18 [3,]  185
19 [4,]  113
20 [5,]   32
21 [6,]  -50
22 [7,]  119
23 [8,]  210
24 > mmul1(c2,a,b)
25 [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,

By the way, the function **clusterApply()** has an optional third argument, used to form the actual

parameter if the function to be applied has two parameters, e.g.

```
> library(parallel)
> c2 <- makeCluster(2)
> f <- function(x,y) x + y
> clusterApply(c2,list(5,12),f,8)
[[1]]
[1] 13

[[2]]
[1] 20
```

A fourth argument can be added if the function has three arguments, and so on.

1.5 Threads Programming in R: Rdsm

As noted, R features the **parallel** package, composed of its old **snow** and **multicore** packages. The former uses message-passing, while the latter involves a weak version of shared-memory access. For those who prefer a general shared-memory interface, there is Rdsm.

R itself is not threaded. However, Rdsm achieves a quasi-thread interface, involving several invocations of R that act as "rtheads," in that (a) the "threads" do operate independently of each other and (b) they genuinely share memory.

This is achieved via operator overloading. All operators in R are actually functions, e.g.

```
> 1+1
[1] 2
> "+"(1,1)
[1] 2
```

including the [operator for array access. What Rdsm does is redefine that operator to access a location in shared memory.

Rdsm is built on top of the packages **snow** and **bigmemory**; the former atter is used for APIs, and for the distribution of shared-memory keys that address memory locations managed by the latter.

1.5.1 Example: Matrix Multiplication

```
# matrix multiplication; the product u %*% v is
# computed on the snow cluster cls, and written
# in-place in w; w is a big.matrix object

mmulthread <- function(u,v,w) {
```

```
    require(parallel)
    # determine which rows this thread will handle
    myidxs <-
       splitIndices(nrow(u),
          myinfo$nwrkrs)[[myinfo$id]]
    # compute this thread's portion of the product
    w[myidxs,] <- u[myidxs,] %*% v[,]
    0  # don't do expensive return of result
}

# test on snow cluster cls
test <- function(cls) {
    # init Rdsm
    mgrinit(cls)
    # set up shared variables a,b,c,
    mgrmakevar(cls,"a",6,2)
    mgrmakevar(cls,"b",2,6)
    mgrmakevar(cls,"c",6,6)
    # fill in some test data
    a[,] <- 1:12
    b[,] <- rep(1,12)
    # give the threads the function to be run
    clusterExport(cls,"mmulthread")
    # run it
    clusterEvalQ(cls,mmulthread(a,b,c))
    print(c[,])  # not print(c)!
}
```

```
> library(parallel)
> c2 <- makeCluster(2)  # 2 threads
> test(c2)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    8    8    8    8    8    8
[2,]   10   10   10   10   10   10
[3,]   12   12   12   12   12   12
[4,]   14   14   14   14   14   14
[5,]   16   16   16   16   16   16
[6,]   18   18   18   18   18   18
```

1.5.2 Example: Maximal Burst in a Time Series

Consider a time series of length n. We may be interested in bursts, periods in which a high average value is sustained. We might stipulate that we look only at periods of length k consecutive points, for a user-specified k. So, we wish to find the period of length k that has the maximal mean value.

Once again, let's leverage the power of R. The **zoo** time series package includes a function **rollmean(w,m)**, which returns all the means of blocks of length k, i.e., what are usually called *moving averages*—just what we need.

Here is the code:

```
# Rdsm code to find max burst in a time series;

# arguments:

#    x:  data vector
#    k:  block size
#    mas:  scratch space, shared, 1 x (length(x)-1)
#    rslts:  2-tuple showing the maximum burst value,
#            and where it starts; shared, 1 x 2

maxburst <- function(x,k,mas,rslts) {
   require(Rdsm)
   require(zoo)
   # determine this thread's chunk of x
   n <- length(x)
   myidxs <- getidxs(n-k+1)
   myfirst <- myidxs[1]
   mylast <- myidxs[length(myidxs)]
   mas[1,myfirst:mylast] <-
      rollmean(x[myfirst:(mylast+k-1)],k)
   # make sure all threads have written to mas
   barr()
   # one thread must do wrapup, say thread 1
   if (myinfo$id == 1) {
      rslts[1,1] <- which.max(mas[,])
      rslts[1,2] <- mas[1,rslts[1,1]]
   }
}

test <- function(cls) {
   require(Rdsm)
   mgrinit(cls)
   mgrmakevar(cls,"mas",1,9)
   mgrmakevar(cls,"rslts",1,2)
   x <<- c(5,7,6,20,4,14,11,12,15,17)
   clusterExport(cls,"maxburst")
   clusterExport(cls,"x")
   clusterEvalQ(cls,maxburst(x,2,mas,rslts))
   print(rslts[,])  # not print(rslts)!
}
```

Chapter 2

Recurring Performance Issues

Oh no! It's actually slower in parallel!—almost everyone's exclamation the first time they try to parallelize code

The available parallel hardware systems sound wonderful at first. But everyone who uses such systems has had the experience of enthusiastically writing his/her first parallel program, anticipating great speedups, only to find that the parallel code actually runs more slowly than the original nonparallel program.

In this chapter, we highlight some major issues that will pop up throughout the book.

2.1 Communication Bottlenecks

Whether you are on a shared-memory, message-passing or other platform, communication is always a potential bottleneck:

- On a shared-memory system, the threads must contend with each other in communicating with memory. And the problem is exacerbated by cache coherency transactions (Section 3.5.1.
- On a cluster, even a very fast network is very slow compared to CPU speeds.
- GPUs are really fast, but their communication with their CPU hosts is slow. There are also memory contention issues as in ordinary shared-memory systems.

Among other things, communication considerations largely drive the load balancing issue, discussed next.

2.2 Load Balancing

Arguably the most central performance issue is **load balancing,** i.e. keeping all the processors busy as much as possible. This issue arises constantly in any discussion of parallel processing.

A nice, easily understandable example is shown in Chapter 7 of the book, *Multicore Application Programming: for Windows, Linux and Oracle Solaris*, Darryl Gove, 2011, Addison-Wesley. There the author shows code to compute the **Mandelbrot set**, defined as follows.

Start with any number c in the complex plane, and initialize z to 0. Then keep applying the transformation

$$z \leftarrow z^2 + c \tag{2.1}$$

If the resulting sequence remains bounded (say after a certain number of iterations), we say that c belongs to the Mandelbrot set.

Gove has a rectangular grid of points in the plane, and wants to determine whether each point is in the set or not; a simple but time-consuming computation is used for this determination.[1]

Gove sets up two threads, one handling all the points in the left half of the grid and the other handling the right half. He finds that the latter thread is very often idle, while the former thread is usually busy—extremely poor **load balance**. We'll return to this issue in Section 2.4.

2.3 "Embarrassingly Parallel" Applications

The term **embarrassingly parallel** is heard often in talk about parallel programming.

2.3.1 What People Mean by "Embarrassingly Parallel"

Consider a matrix multiplication application, for instance, in which we compute AX for a matrix A and a vector X. One way to parallelize this problem would be to have each processor handle a group of rows of A, multiplying each by X in parallel with the other processors, which are handling other groups of rows. We call the problem embarrassingly parallel, with the word "embarrassing" meaning that the problem is too easy, i.e. there is no intellectual challenge involved. It is pretty obvious that the computation Y = AX can be parallelized very easily by splitting the rows of A into groups.

[1] You can download Gove's code from `http://blogs.sun.com/d/resource/map_src.tar.bz2`. Most relevant is **listing7.64.c**.

By contrast, most parallel sorting algorithms require a great deal of interaction. For instance, consider Mergesort. It breaks the vector to be sorted into two (or more) independent parts, say the left half and right half, which are then sorted in parallel by two processes. So far, this is embarrassingly parallel, at least after the vector is broken in half. But then the two sorted halves must be merged to produce the sorted version of the original vector, and that process is *not* embarrassingly parallel; it can be parallelized, but in a more complex, less obvious manner.

Of course, it's no shame to have an embarrassingly parallel problem! On the contrary, except for showoff academics, having an embarrassingly parallel application is a cause for celebration, as it is easy to program.

In recent years, the term **embarrassingly parallel** has drifted to a somewhat different meaning. Algorithms that are embarrassingly parallel in the above sense of simplicity tend to have very low communication between processes, key to good performance. That latter trait is the center of attention nowadays, so the term **embarrassingly parallel** generally refers to an algorithm with low communication needs.

For that reason, many people would NOT considered even our prime finder example in Section 1.4.3 to be embarrassingly parallel. Yes, it was embarrassingly easy to write, but it has high communication costs, as both its locks and its global array are accessed quite often.

On the other hand, the Mandelbrot computation described in Section 2.2 is truly embarrassingly parallel, in both the old and new sense of the term. There the author Gove just assigned the points on the left to one thread and the rest to the other thread—very simple—and there was no communication between them.

2.3.2 Iterative Algorithms

Many parallel algorithms involve iteration, with a rendezvous of the tasks after each iteration. Within each iteration, the nodes act entirely independently of each other, which makes the problem seem embarrassingly parallel.

But unless the **granularity** of the problem is coarse, i.e. there is a large amount of work to do in each iteration, the communication overhead will be significant, and the algorithm may not be considered embarrassingly parallel.

2.4 Static (But Possibly Random) Task Assignment Typically Better Than Dynamic

Say an algorithm generates t independent[2] tasks and we have p processors to handle them. In our matrix-times-vector example of Section 1.4.1, say, each row of the matrix might be considered one task. A processor's work would then be to multiply the vector by this processor's assigned rows of the matrix.

How do we decide which tasks should be done by which processors? In **static** assignment, our code would decide at the outset which processors will handle which tasks. The alternative, **dynamic** assignment, would have processors determine their tasks as the computation proceeds.

In the matrix-times-vector example, say we have 10000 rows and 10 processors. In static task assignment, we could pre-assign processor 0 rows 0-999, processor 1 rows 1000-1999 and so on. On the other hand, we could set up a **task farm**, a queue consisting here of the numbers 0-9999. Each time a processor finished handling one row, it would remove the number at the head of the queue, and then process the row with that index.

It would at first seem that dynamic assignment is more efficient, as it is more flexible. However, accessing the task farm, for instance, entails communication costs, which might be very heavy. In this section, we will show that it's typically better to use the static approach, though possibly randomized.[3]

2.4.1 Example: Matrix-Vector Multiply

Consider again the problem of multiplying a vector X by a large matrix A, yielding a vector Y. Say A has 10000 rows and we have 10 threads. Let's look at little closer at the static/dynamic tradeoff outlined above. For concreteness, assume the shared-memory setting.

There are several possibilities here:

- **Method A:** We could simply divide the 10000 rows into chunks of 10000/10 = 1000, and parcel them out to the threads. We would pre-assign thread 0 to work on rows 0-999 of A, thread 1 to work on rows 1000-1999 and so on.

  This is essentially OpenMP's **static** scheduling policy, with default chunk size.[4]

  There would be no communication between the threads this way, but there could be a problem of load imbalance. Say for instance that by chance thread 3 finishes well before the others.

[2] Note the qualifying term.

[3] This is still static, as the randomization is done at the outset, before starting computation.

[4] See Section 4.3.3.

Then it will be idle, as all the work had been pre-allocated.

- **Method B:**

  OpenMP's **dynamic** policy does what the name implies, which can be described as follows (in non-OpenMP terms):

  We would have a shared variable named,say, **nextchunk** similar to **nextbase** in our prime-finding program in Section 1.4.3. Each time a thread would finish a chunk, it would obtain a new chunk to work on, by recording the value of **nextchunk** and incrementing that variable by 1 (all atomically, of course).

  This approach would have better load balance, because the first thread to find there is no work left to do would be idle for at most 100 rows' amount of computation time, rather than 1000 as above. Meanwhile, though, communication would increase, as access to the locks around **nextchunk** would often make one thread wait for another.[5]

- **Method C:** So, Method A above minimizes communication at the possible expense of load balance, while the Method B does the opposite.

OpenMP also offers the **guided** policy, which is like **dynamic** except the chunk size decreases over time.

I will now show that in typical settings, the Method A above (or a slight modification) works well. To this end, consider a chunk consisting of m tasks, such as m rows in our matrix example above, with times $T_1, T_2, ..., T_m$. The total time needed to process the chunk is then $T_1 + ..., T_m$.

The T_i can be considered random variables; some tasks take a long time to perform, some take a short time, and so on. As an idealized model, let's treat them as independent and identically distributed random variables. Under that assumption (if you don't have the probability background, follow as best you can), we have that the mean (**expected value**) and variance of total task time are

$$E(T_1 + ..., T_m) = mE(T_1)$$

and

$$Var(T_1 + ..., T_m) = mVar(T_1)$$

[5]Why are we calling it "communication" here? Recall that in shared-memory programming, the threads communicate through shared variables. When one thread increments **nextchunk**, it "communicates" that new value to the other threads by placing it in shared memory where they will see it, and as noted earlier contention among threads to shared memory is a major source of potential slowdown.

Thus

$$\frac{\text{standard deviation of chunk time}}{\text{mean of chunk time}} \sim O\left(\frac{1}{\sqrt{m}}\right)$$

In other words:

- run time for a chunk is essentially constant if m is large, and
- there is essentially no load imbalance in Method A

Since load imbalance was the only drawback to Method A and we now see it's not a problem after all, then Method A is best.

For more details and timing examples, see N. Matloff, "Efficient Parallel R Loops on Long-Latency Platforms," *Proceedings of the 42nd Interface between Statistics and Computer Science*, Rice University, June 2012.[6]

2.4.2 Load Balance, Revisited

But what about the assumptions behind that reasoning? Consider for example the Mandelbrot problem in Section 2.2. There were two threads, thus two chunks, with the tasks for a given chunk being computations for all the points in the chunk's assigned region of the picture.

Gove noted there was fairly strong load imbalance here, and that the *reason* was that most of the Mandelbrot points turned out to be in the left half of the picture! The computation for a given point is iterative, and if a point is not in the set, it tends to take only a few iterations to discover this. That's why the thread handling the right half of the picture was idle so often.

So Method A would not work well here, and upon reflection one can see that the problem was that the tasks within a chunk were not independent, but were instead highly correlated, thus violating our mathematical assumptions above. Of course, before doing the computation, Gove didn't know that it would turn out that most of the set would be in the left half of the picture. But, one could certainly anticipate the correlated nature of the points; if one point is not in the Mandelbrot set, its near neighbors are probably not in it either.

But Method A can still be made to work well, via a simple modification: Simply form the chunks randomly. In the matrix-multiply example above, with 10000 rows and chunk size 1000, do NOT

[6] As noted in the Preface to this book, I occasionally refer here to my research, to illustrate for students the beneficial interaction between teaching and research.

assign the chunks contiguously. Instead, generate a random permutation of the numbers 0,1,...,9999, naming them $i_0, i_1, ..., i_{9999}$. Then assign thread 0 rows $i_0 - i_{999}$, thread 1 rows $i_{1000} - i_{1999}$, etc.

In the Mandelbrot example, we could randomly assign rows of the picture, in the same way, and avoid load imbalance.

So, actually, Method A, or let's call it Method A', will still typically work well.

2.4.3 Example: Mutual Web Outlinks

Here's an example that we'll use at various points in this book:

> **Mutual outlinks in a graph:**
>
> 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. Say we want to find the number of mutual outlinks, averaged over all pairs of Web sites.
>
> Let A be the **adjacency matrix** of the graph. Then the mean of interest would be found as follows:

```
1  sum = 0
2  for i = 0...n-2
3      for j = i+1...n-1
4          count = 0
5          for k = 0...n-1 count += a[i][k] * a[j][k]
6  mean = sum / (n*(n-1)/2)
```

> Say again n = 10000 and we have 10 threads. We should not simply assign work to the threads by dividing up the i loop, with thread 0 taking the cases i = 0,...,999, thread 1 the cases 1000,...,1999 and so on. This would give us a real load balance problem. Thread 8 would have much less work to do than thread 3, say.
>
> We could randomize as discussed earlier, but there is a much better solution: Just pair the rows of A. Thread 0 would handle rows 0,...,499 and 9500,...,9999, thread 1 would handle rows 500,999 and 9000,...,9499 etc. This approach is taken in our OpenMP implementation, Section 4.12.
>
> In other words, Method A still works well.

In the mutual outlinks problem, we have a good idea beforehand as to how much time each task needs, but this may not be true in general. An alternative would be to do *random* pre-assignment of tasks to processors.

On the other hand, if we know beforehand that all of the tasks should take about the same time, we should use static scheduling, as it might yield better cache and virtual memory performance.

2.4.4 Work Stealing

There is another variation to Method A that is of interest today, called **work stealing**. Here a thread that finishes its assigned work and has thus no work left to do will "raid" the work queue of some other thread. This is the approach taken, for example, by the elegant Cilk language. Needless to say, accessing the other work queue is going to be expensive in terms of time and memory contention overhead.

2.4.5 Timing Example

I ran the Mandelbrot example on a shared memory machine with four cores, two threads per core, with the following results for eight threads, on an 8000x8000 grid:

policy	time
static	47.8
dynamic	21.4
guided	29.6
random	15.7

Default values were used for chunk size in the first three cases. I did try other chunk sizes for the **dynamic** policy, but it didn't make much difference. See Section 4.4 for the code.

Needless to say, one shouldn't overly extrapolate from the above timings, but it does illustrate the issues.

2.5 Latency and Bandwidth

We've been speaking of communications delays so far as being monolithic, but they are actually (at least) two-dimensional. The key measures are **latency** and **bandwidth**:

- Latency is the time it takes for one bit to travel for source to destination, e.g. from a CPU to memory in a shared memory system, or from one computer to another in a cluster.
- Bandwidth is the number of bits per unit time that can be input into the communications channel. This can be affected by factors such as bus width in a shared memory system and

number of parallel network paths in a message passing system, and also by the speed of the links.

It's helpful to think of a bridge, with toll booths at its entrance. Latency is the time needed for one car to get from one end of the bridge to the other. Bandwidth is the number of cars that can enter the bridge per unit time. We can reduce latency by increasing the speed limit, and can increase bandwidth by improving the speed by which toll takers can collect tolls, and increasing the number of toll booths.

Latency hiding:

One way of dealing with long latencies is known as **latency hiding**. The idea is to do a long-latency operation in parallel with something else.

For example, GPUs tend to have very long memory access times, but this is solved by having many pending memory accesses at the same time. During the latency of some accesses, earlier ones that have now completed can now be acted upon (Section 5.3.3.2).

2.6 Relative Merits: Performance of Shared-Memory Vs. Message-Passing

My own preference is shared-memory, but there are pros and cons to each paradigm.

It is generally believed in the parallel processing community that the shared-memory paradigm produces code that is easier to write, debug and maintain than message-passing. See for instance R. Chandra, *Parallel Programming in OpenMP*, MKP, 2001, pp.10ff (especially Table 1.1), and M. Hess *et al*, Experiences Using OpenMP Based on Compiler Directive Software DSM on a PC Cluster, in *OpenMP Shared Memory Parallel Programming: International Workshop on OpenMP Applications and Tools*, Michael Voss (ed.), Springer, 2003, p.216.

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

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

As will be seen, though, GPU is hardly a panacea. Where, then, are people to get access to large-

scale parallel systems? Most people do not (currently) have access to large-scale multicore machines, while most do have access to large-scale message-passing machines, say in cloud computing venues. Thus message-passing plays a role even for those of us who preferred the shared-memory paradigm.

Also, hybrid systems are common, in which a number of shared-memory systems are tied together by, say, MPI.

2.7 Memory Allocation Issues

Many algorithms require large amounts of memory for intermediate storage of data. It may be prohibitive to allocate this memory statically, i.e. at compile time. Yet dynamic allocation, say via **malloc()** or C++'s **new** (which probably produces a call to **malloc() anyway**, is very expensive in time.

Using large amounts of memory also can be a major source of overhead due to cache misses and page faults.

One way to avoid **malloc()**, of course, is to set up static arrays whenever possible.

There are no magic solutions here. One must simply be aware of the problem, and tweak one's code accordingly, say by adjusting calls to **malloc()** so that one achieves a balance between allocating too much memory and making too many calls.

2.8 Issues Particular to Shared-Memory Systems

This topic is covered in detail in Chapter 3, but is so important that the main points should be mentioned here.

- Memory is typically divided into **banks**. If more than one thread attempts to access the same bank at the same time, that effectively serializes the program.
- There is typically a cache at each processor. Keeping the contents of these caches consistent with each other, and with the memory itself, adds a lot of overhead, causing slowdown.

In both cases, awareness of these issues should impact how you write your code.

See Sections 3.2 and 3.5.

Chapter 3

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 A.3.1.

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

```
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

```
1  parallel for thr = 0 to 15
2     localsum = 0
3     for j = 0 to 999999
4        localsum += x[thr*1000000+j]
5     grandsum += localsum  // critical section
```

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:

```
1  parallel for thr = 0 to 15
2     localsum = 0
3     for j = 0 to 999999
4        localsum += x[16*j+thr]
5     grandsum += localsum
```

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.
- 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.

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

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:

```
1 struct {
2    char name[25];
3    float salary;
4    float yrs;
5 } 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:

```
1 struct {
2    char name[100];
3    float salary[100];
4    float yrs[100];
5 }
```

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.

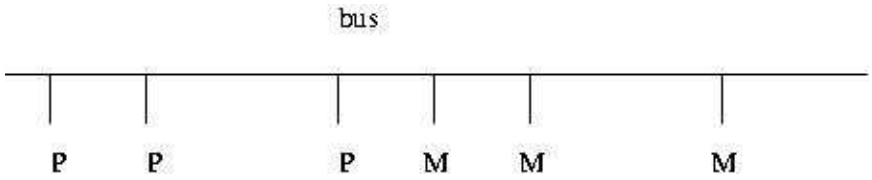
```
1
2 // routines to initialize, read and write
3 // padded versions of a matrix of floats;
4 // the matrix is nominally mxn, but its
5 // rows will be padded on the right ends,
6 // so as to enable a stride of s down each
7 // column; it is assumed that s >= n
```

```
 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.