One way of reducing the overhead arising from the network system software is to use *remote direct memory access* (RDMA), which involves both nonstandard hardware and software. The name derives from the Direct Memory Acess devices that are common in even personal computers today.

When reading from a fast disk, for instance, DMA bypasses the "middleman," the CPU, and writes directly to memory, a significant speedup. (DMA devices in fact are special-purpose CPUs in their own right, designed to copy data directly between an input-output device and memory.) Disk writes are made faster the same way.

With RDMA, we bypass a different kind of middleman, in this case the network protocol stack. When reading a message arriving from the network, RDMA deposits the message directly into the memory used by our program.

4.2 Rmpi

As noted, **Rmpi** is an R interface to the famous MPI protocol, the latter normally being accessed via C, C++ or FORTRAN. MPI consists of hundreds of functions callable from user programs.

Note that MPI also provides network services beyond simply sending and receiving messages. An important point is that it enforces message order. If say, messages A and B are sent from process 8 to process 3 in that order, then the program at process 3 will receive them in that order. A call at process 3 to receive from process 8 will receive A first, with B not being processed until the second such call.[1] This makes the logic in your application code much easier to write.

In addition, MPI allows the programmer to define several different kinds of messages. One might make a call, for instance, that says in essence, "read the next message of type 2 from process 8," or even "read the next message of type 2 from any process."

Rmpi provides the R programmer with access to such operations, and also provides some new R-specific messaging operations.

With all that power comes complexity. **Rmpi** can be tricky to install—and even to launch—with various platform dependencies to deal with, even in terms of how the manager launches the workers. These issues, as well as the plethora of functions available in **Rmpi** and the plethora of options in those functions, are beyond the scope of this book. Instead, the hope here

[1]This assumes that the calls do not specify message type, discussed below.

is to present a good introduction to the message-passing paradigm, with **Rmpi** as our vehicle.

4.3 Example: Pipelined Prime Number Finder

Prime numbers play a key role (pun intended) in cryptography, the core of data security. R may not be the best vehicle for finding them, but this does make for an easy-to-understand example of **Rmpi** and the message-passing paradigm, and thus is used here.

Again in the interest of simplicity, the code is not intended to be optimal. In fact, the nonoptimality will serve as a springboard for discussion of typical performance issues that arise with message-passing systems.

4.3.1 The Code

```
# Rmpi code to find prime numbers

# for illustration purposes, not intended to be optimal

# returns vector of all primes in 2..n; the vector "divisors" is used as
# a basis for a Sieve of Erathosthenes operation; must have n <=
# (max(divisors)^2) and n even

# the argument "msgsize" controls the chunk size in communication from
# the manager to the first worker, node 1

primepipe <- function(n,divisors,msgsize) {
   mpi.bcast.Robj2slave(dowork)
   mpi.bcast.Robj2slave(dosieve)
   # start workers; note nonblocking call
   mpi.bcast.cmd(dowork,n,divisors,msgsize)
   # remove the evens right away
   odds <- seq(from=3,to=n,by=2)
   nodd <- length(odds)
   # send odds to node 1, in chunks
   startmsg <- seq(from=1,to=nodd,by=msgsize)
   for (s in startmsg) {
      rng <- s:min(s+msgsize-1,nodd)
      mpi.send.Robj(odds[rng],tag=0,dest=1)
   }
   # send end-data sentinel
   mpi.send.Robj(NA,tag=0,dest=1)
   # receive results from last node
   lastnode <- mpi.comm.size()-1
   # return te result; don't forget the 2
   c(2,mpi.recv.Robj(tag=0,source=lastnode))
}

# worker code
dowork <- function(n,divisors,msgsize) {
   # which are my divisors?
   me <- mpi.comm.rank()
   lastnode <- mpi.comm.size()-1
   ld <- length(divisors)
   tmp <- floor(ld / lastnode)
   mystart <- (me-1) * tmp + 1
   myend <- mystart + tmp - 1
   if (me == lastnode) myend <- ld
   mydivs <- divisors[mystart:myend]
   if (me == lastnode) out <- NULL
   repeat {
      msg <- mpi.recv.Robj(tag=0,source=me-1)
      if (me < lastnode) {
         if (!is.na(msg[1])) {
            sieveout <- dosieve(msg,mydivs)
            mpi.send.Robj(sieveout,tag=0,dest=me+1)
         } else {  # no more coming in, so send sentinel
            mpi.send.Robj(NA,tag=0,dest=me+1)
            return()
         }
      } else {
         if (!is.na(msg[1])) {
            sieveout <- dosieve(msg,mydivs)
            out <- c(out,sieveout)
         } else {  # no more coming in, so send results to manager
            mpi.send.Robj(out,tag=0,dest=0)
            return()
         }
      }
   }
}

# check divisibility of the current chunk x
```

```r
dosieve <- function(x,divs) {
   for (d in divs) {
      x <- x[x %% d != 0 | x == d]
   }
   x
}

# serial prime finder; can be used to generate divisor list of primepipe
serprime <- function(n) {
   nums <- 1:n
   x <- rep(1,n)
   maxdiv <- ceiling(sqrt(n))
   for (d in 2:maxdiv) {
      if (x[d])
         x[x !=0 & nums > d & nums %% d == 0] <- 0
   }
   nums[x != 0 & nums >= 2]
}
```

4.3.2 Usage

The function **primepipe** has three arguments:

- **n**: the function returns the vector of all primes between 2 and **n**, inclusive
- **divisors**: the function checks each potential prime for divisibility by the numbers in this vector
- **msgsize**: the size of messages from the manager to the first worker

Here are the details:

This is the classical Sieve of Eratosthenes. We make a list of the numbers from 2 to n, then "cross out" all multiples of 2, all multiples of 3 and so on. After the crossing-out by 2s and 3s, for instance, our list will look like this:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ ...

In the end, the numbers that haven't gotten crossed out are the primes.

The vector **divisors** "primes the pump," as it were. We find a small set of primes using nonparallel means, and then use those in the larger parallel problem. But what range do we need for them? Reason as follows.

If a number i has a divisor k larger than $\sqrt{n}$, it must then have one (specifically, the number i/k) smaller than that value. Thus in crossing out all multiples of k, we need only consider values of k up to $\sqrt{n}$. So, in order to achieve our goal of finding all the primes up through n, we take our **divisors** vector to be all the primes up through $\sqrt{n}$.

The function **serprime()** in the code above will do that. For example, say n is 1000. Then we first find all the primes less than or equal to $\sqrt{1000}$, using our nonparallel function:

```
> dvs <- serprime(ceiling(sqrt(1000)))
> dvs
 [1]  2  3  5  7 11 13 17 19 23 29 31
```

We then use these numbers in our parallel function to find the primes up through 1000:

```
> primepipe(1000,dvs,100)
 [1]   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53
59 61
 [19]  67  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151
 [37] 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251
 [55] 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359
 [73] 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463
 [91] 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593
[109] 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701
[127] 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827
[145] 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953
[163] 967 971 977 983 991 997
```

Now, to understand the argument **msgsize**, consider the case n = 1000 above. Each worker will be responsible for its particular chunk of **divisors**. If we have two workers, then Process 0 (the manager) will "cross out" multiples of 2; process 1 (the first worker) will handle multiples of 3, 5, 7, 11 and 13; process 2 will handle k = 17, 19, 23, 29 and 31. So, process 0 will cross out the multiples of 2, and send the remaining numbers, the odds, to process 1. The argument **msgsize** specifies the size of chunks of odds that process 0 sends to process 1. More on this point later.

4.3.3 Timing Example

Let's try the case n = 10000000. The serial code took time 424.592 seconds.

Let's try it in parallel on a network of PCs, for first two, then three and then four workers. with various values for **msgsize**. The results are shown in Table 4.1.

msgsize	2 workers	3 workers	4 workers
1000	59.487	58.175	47.248
5000	22.855	17.541	15.454
10000	19.230	14.734	12.522
15000	19.198	14.874	12.689
25000	22.516	18.057	15.591
50000	23.029	18.573	16.114

Table 4.1: Timings, Prime Number Finding

The parallel version was indeed faster than the serial one. This was partly due to parallelism and partly to the fact that the parallel version is more efficient, since the serial algorithm does more total crossouts. A fairer comparison might be a recursive version of **serprime()**, which would reduce the number of crossouts. But there are other important facets of the timing numbers.

First, as expected, using more workers produced more speed, at least in the range tried here. Note, though, that the speedup was not linear. The best time for three workers was only 30% better than that for two workers, compared to a "perfect" speedup of 50%. Using four workers instead of two yields only a 53% gain. We'll return to this point shortly.

4.3.4 Latency, Bandwdith and Parallelism

Another salient aspect here is that **msgsize** matters. Recall Section 2.2.6, especially Equation (2.1). Let's see how they affect things here.

In our timings above, setting the **msgsize** parameter to the lower value, 1000, results in have more chunks, thus more times that we incur the network latency l. On the other hand, a value of 50000 less parallelism, and impedes are ability to engage in latency hiding (Section 2.2.6), in which we try to overlap computation and communication; this reduces parallelism and thus reduces speed.