CHAPTER 13

Advanced Point-to-Point Communication

UP UNTIL NOW THE ONLY POINT-TO-POINT communication functions we've used are the blocking operations `MPI_Send` and `MPI_Recv`. In this chapter, we'll explore the much richer set of point-to-point functions provided by MPI. We'll begin by developing a couple of simple implementations of an allgather function using `MPI_Send` and `MPI_Recv`. With this under our belts, we'll spend the remainder of the chapter discussing how we might modify our functions to make them more reliable and faster.

The first new communication functions we'll discuss are `MPI_Sendrecv` and `MPI_Sendrecv_replace`, which provide a simple means of organizing paired communications to avoid deadlock. We'll continue with a discussion of the basic nonblocking functions, `MPI_Isend` and `MPI_Irecv`. As we'll see, these nonblocking functions provide a means of overlapping communication and computation, and hence a means of improving performance. We'll go on to a discussion of persistent communication requests, which provide the possibility of even greater overlap between communication and computation. We'll close with a discussion of communication modes: up to now, we've left a number of decisions about how communication is managed to the system. For example, we've let the system manage buffering of messages. Using different communication modes, we can explicitly tell the system whether we want messages buffered and, if so, where they should be buffered.

13.1 An Example: Coding Allgather

MPI provides a collective communication operation that allows us to take data that is distributed across a collection of processes and gather the distributed data onto each process. The name of the MPI function that does this is `MPI_Allgather`. As an example, let's implement our own allgather functions, first using `MPI_Send` and `MPI_Recv`, then using more advanced point-to-point communications.

There are *many* possible implementations of allgather in terms of point-to-point operations. If our processes were physically configured as a full binary tree, a natural solution would be to write a tree-structured gather to the root process, and then follow that by a tree-structured broadcast from the root. The cost of the startups for this would be $2(\lceil \log_2(p) \rceil - 1)t_s$.

If our processors were physically configured as a ring, a **ring pass** would be a natural solution. The idea is that each process sends its data to the process immediately adjacent in, say, the counterclockwise direction. Then this operation can be repeated, but instead of using the original data, each process sends the data it received during the previous stage. Figure 13.1 shows a four-processor ring pass. Since it takes exactly $p-1$ stages for each processor to receive all the data, the cost of the startups is $(p-1)t_s$.

An alternative that is natural for hypercubes involves a sequence of pairwise exchanges among the processes. At each stage, the processes are split into two groups of equal sizes, each process in the first group is paired with a process in the second, and the paired processes exchange *all* of their data. See Figure 13.2 for an example with four processes. If there are $p = 2^d$ processes, then at each stage each process will double the amount of data it contains. Hence there will be $d = \log_2(p)$ stages and $\log_2(p)$ startups.

Let's write a ring pass allgather and a hypercube allgather.

13.1.1 Function Parameters

We'll assume that the data to be gathered from each process consists of a block of floats, and each process is storing the same number of floats. Thus the input parameters should be an array of floats containing the data to be gathered from the local process, the size of the array (which will be the same on each process), and a communicator whose members are the processes that will participate in the allgather. The output will consist of a single parameter: an array containing the floats that have been gathered from each process. Thus, the output array will have order pb, where p is the size of the communicator and b is the order of the input array. As with the MPI collective functions, we will assume that the input and output arrays are distinct. This will mean that in each function we will need to copy the contents of the input array into the appropriate location in the output array.

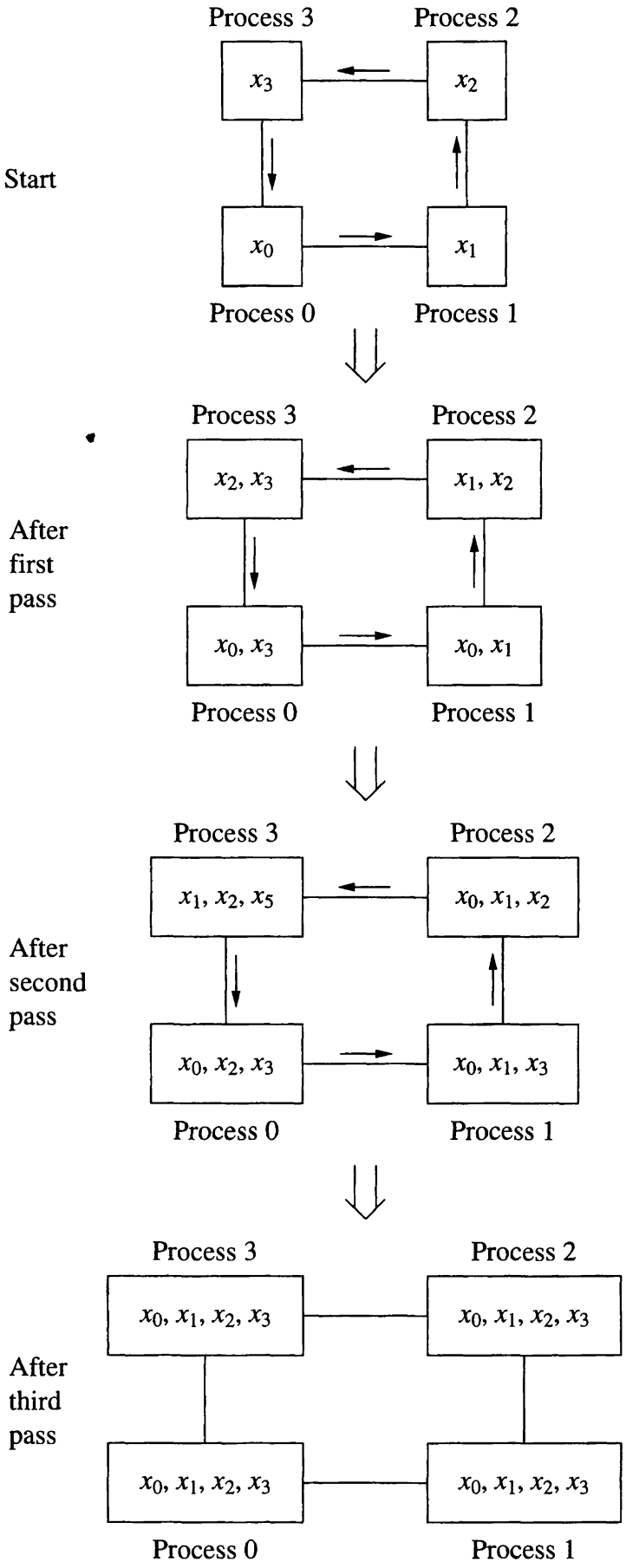

Figure 13.1 A four-processor ring pass

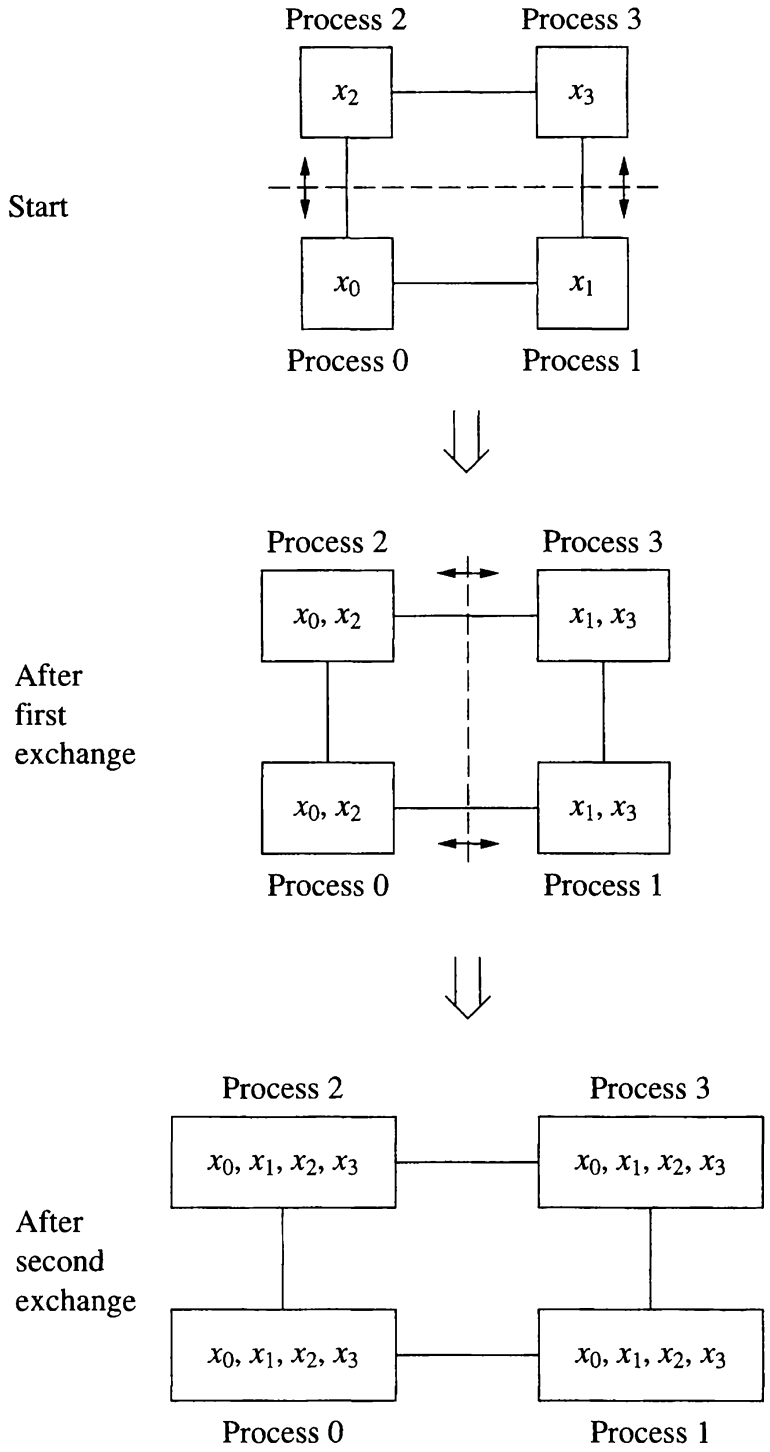

Figure 13.2 Four-process pairwise exchange

13.1.2 Ring Pass Allgather

The heart of the code is a loop: during each pass through the loop we send the most recently received block of data to the successor processor and receive a new block of data from the predecessor processor. Thus, we will execute the body of the loop $p - 1$ times; before we execute the loop, we need to compute the ranks of the predecessor and successor processes, as well as copying the data from the input array into the output array.

In the body of the loop we will need to compute offsets into the array that tell us where the data to be sent begin and where the data to be received should begin. This involves some modular arithmetic: we work backward through the array one block at a time; after we've hit the beginning of the array, we move to the end of the array. The block we're receiving will always be one block before the block we're sending. Since we start with our own block, the block we send on pass i, $i = 0, 1, \ldots, p-1$, will be

$$(\texttt{my\_rank} - i) \bmod p.$$

Since the C programming language does not guarantee that the result of `a % b` is positive, we should add in p to the dividend. That is, the block that we should send on the ith pass can be computed in C as

```
(my_rank - i + p) % p
```

In order to convert this to the offset in floats instead of blocks, we can multiply by the blocksize

```
((my_rank - i + p) % p)*blocksize
```

The block that we're receiving will be the block immediately preceding the one we're sending. So to get the offset in floats for the receive, we simply subtract one from the dividend:

```
((my_rank - i + p - 1) % p)*blocksize
```

Then we execute our send and receive.

```
void Allgather_ring(
          float     x[]          /* in  */,
          int       blocksize    /* in  */,
          float     y[]          /* out */,
          MPI_Comm  ring_comm    /* in  */) {

    int         i, p, my_rank;
    int         successor, predecessor;
    int         send_offset, recv_offset;
    MPI_Status  status;

    MPI_Comm_size(ring_comm, &p);
    MPI_Comm_rank(ring_comm, &my_rank);

    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];

    successor = (my_rank + 1) % p;
```

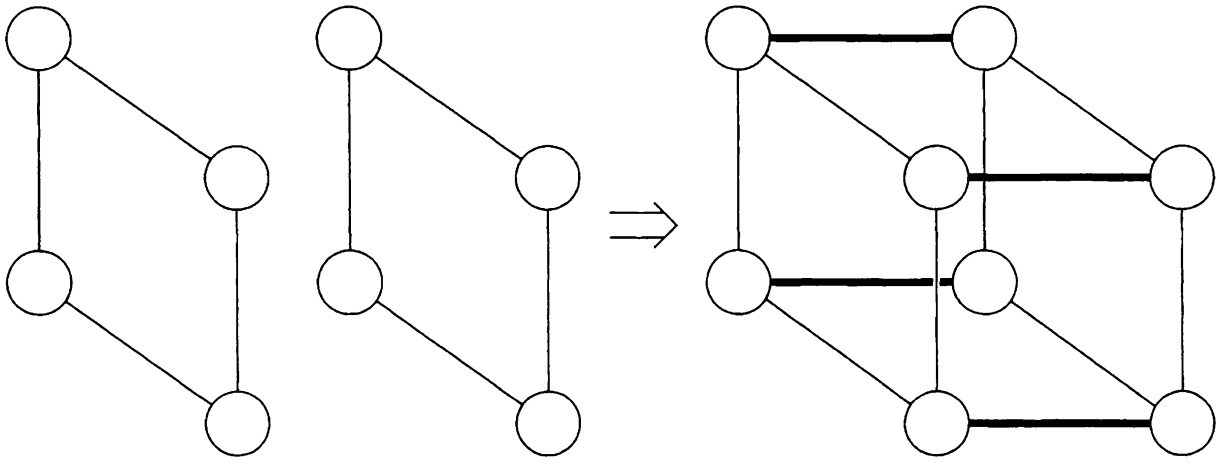

Figure 13.3 Forming a three-dimensional hypercube from two two-dimensional hypercubes

```
    predecessor = (my_rank - 1 + p) % p;

    for (i = 0; i < p - 1; i++) {
        send_offset = ((my_rank - i + p) % p)*blocksize;
        recv_offset =
            ((my_rank - i - 1 + p) % p)*blocksize;
        MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
            successor, 0, ring_comm);
        MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
            predecessor, 0, ring_comm, &status);
    }
} /* Allgather_ring */
```

Note that we execute all the sends first and then all the receives. This assumes that the system can buffer the messages: indeed, the code could fail if there is inadequate system buffering. We'll return to this point later.

13.2 Hypercubes

For the hypercube allgather, we'll assume that p is a power of two, so that we don't have to worry about how to deal with processes that aren't paired up. As with most hypercube algorithms, it's usually easiest to understand how the algorithm works by looking at the binary representation of the process ranks.

Recall that hypercubes are defined inductively: a hypercube of dimension 0 consists of a single process, and a hypercube of dimension $d + 1$ can be constructed from two hypercubes of dimension d by joining corresponding processes in the d-dimensional hypercubes with communication wires. See Figure 13.3.

This "doubling" process leads to a natural scheme for assigning binary addresses to the processes: each process in a d-dimensional hypercube has a

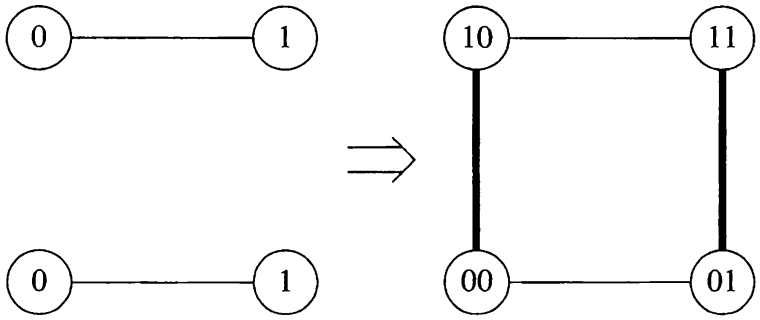

Figure 13.4 Addresses in a two-dimensional hypercube built from two one-dimensional hypercubes

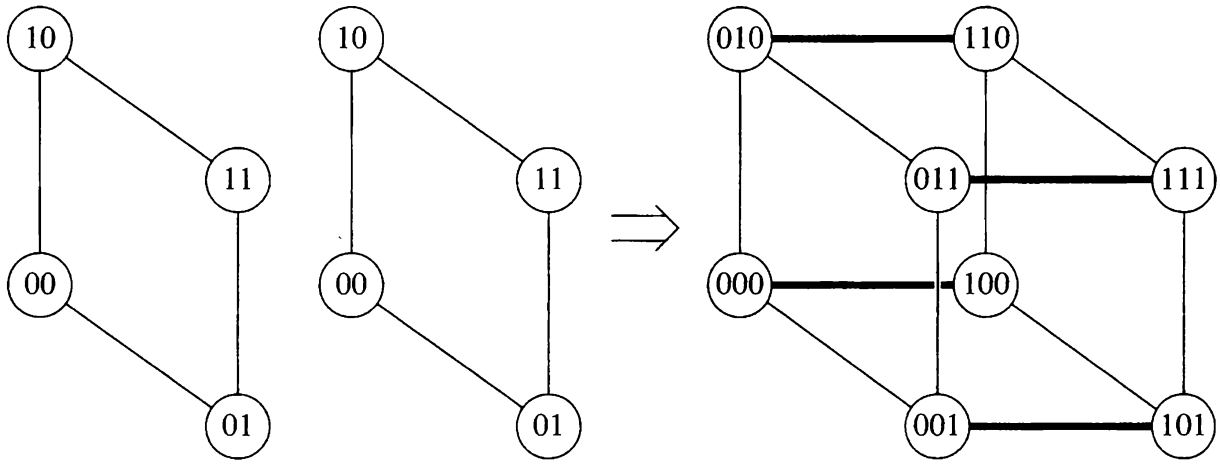

Figure 13.5 Addresses in a three-dimensional hypercube built from two two-dimensional hypercubes

unique d-bit address. (What happens in a zero-dimensional hypercube?) In a one-dimensional hypercube, one process has rank 0 and the other has rank 1. Now if we form a two-dimensional hypercube from two one-dimensional hypercubes, we'll join the two processes with rank 0 with a wire and we'll join the two processes with rank 1 with a wire. In order to get unique addresses, we'll use 2-bit addresses in our new hypercube: the low-order bits will be the same as they were in the original one-dimensional hypercubes, but the high-order bit will be 0 in one of the original one-dimensional hypercubes and 1 in the other one-dimensional hypercube. See Figure 13.4.

In order to assign addresses in a three-dimensional hypercube, we use the same idea that we used to assign addresses in the two-dimensional hypercube: after we join the two two-dimensional hypercubes, the two low-order bits of the address are the same as they were in the original hypercubes, but the high-order bit is 0 for all the processes in one of the original two-dimensional hypercubes and the high-order bit is 1 for all the processes in the other. See Figure 13.5.

The idea generalizes to arbitrary dimensions, and it results in an addressing scheme that has the following property:

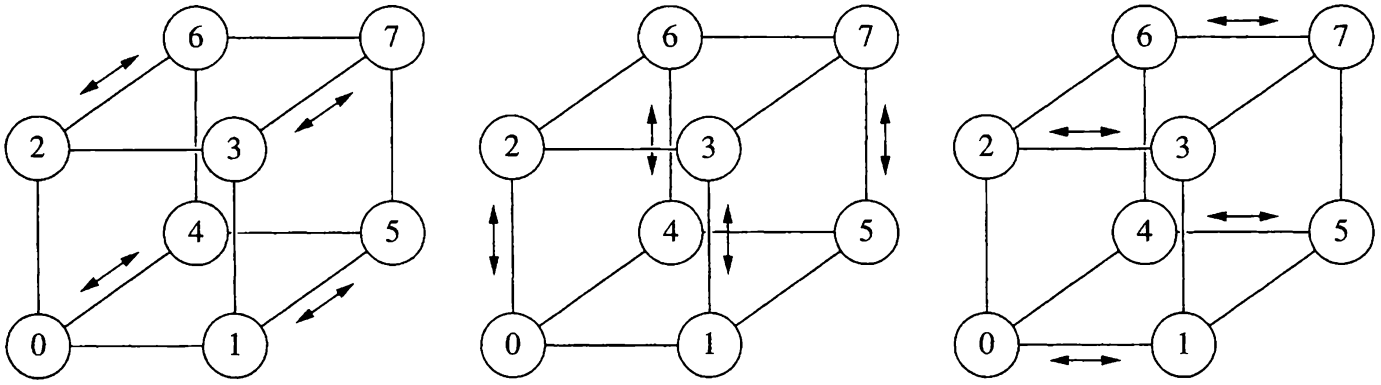

Figure 13.6 Process pairing during the stages

Two processes in a hypercube are adjacent if and only if their binary addresses differ in exactly one bit.

Now we can formulate our pairing scheme in allgather more precisely. Suppose the hypercube has dimension d. During the first stage, we'll assign all the processes that have 0 as their most significant bit to one group, and all the processes that have 1 as their most significant bit to the other group. Each process in the first group will be paired with the process in the second group whose binary address differs only in the most significant bit. Then each communication will occur between adjacent processes, and no two pairs of communicating processes will attempt to use the same communication wire. During the next stage, we'll use the same idea, except that now we'll use the second most significant bit. That is, the first group will now consist of all processes with 0 as their second most significant bit, and the second group will consist of all processes with 1 as their second most significant bit. We'll pair processes whose binary addresses are identical in all but the second most significant bit. Continuing this process: next we use the third most significant bit, then the fourth, etc., until we've formed the groups and pairs on the least significant bit.

An example always helps. Suppose that the blocksize is 1, and we have a three-dimensional cube. Then the stages are illustrated in Figures 13.6–13.9.

13.2.1 Additional Issues in the Hypercube Exchange

At each stage, each process will need to compute the rank of the process with which it will exchange data. But observe that this won't be enough: the amount of data to be transmitted and its layout change with each stage. In our example, during the first stage, each process simply sends and receives a single float. During the second stage, each process sends and receives two floats that are spaced four floats apart, and during the third stage, each process sends and receives four floats that are spaced two floats apart. In general, we will need

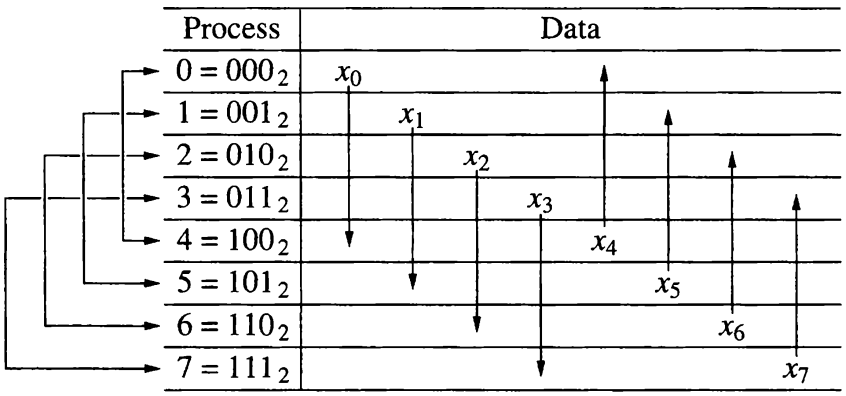

Figure 13.7 Exchanges during stage 0

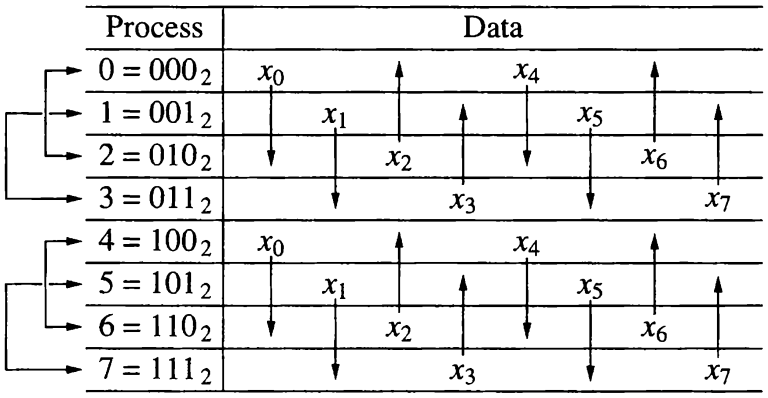

Figure 13.8 Exchanges during stage 1

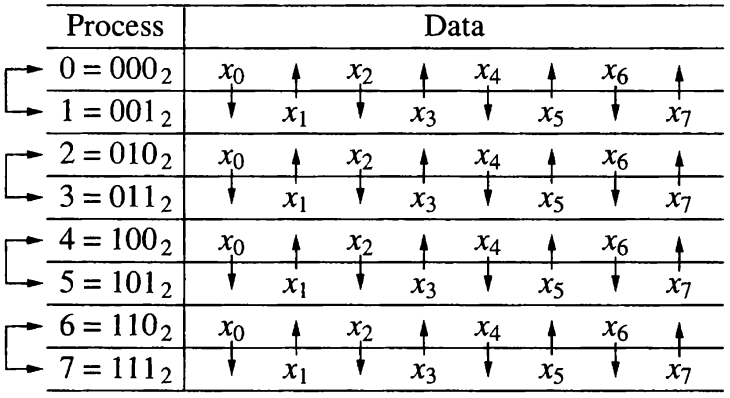

Figure 13.9 Exchanges during stage 2

to use blocks of floats instead of simple floats, but the problem is essentially the same.

There are a number of approaches to dealing with this. We list a few possibilities:

1. We can build a new derived datatype before each exchange.
2. We can pack the data before each exchange and unpack it afterward.

3. We can send/receive the data into contiguous memory locations at every stage and sort the data when we're done.
4. We can work under the assumption that the function will be called many times with the same blocksize and prebuild all the derived types we'll need.

The fourth approach is clearly less general than the preceding three. Is there a clear reason for choosing among the first three? Probably, but it will depend on your system and the problem size. For example, if your system has a very efficient implementation of derived types, then the first alternative will probably be faster than the second, especially if the number of processes and the blocksize are very large: the use of pack and unpack will involve a lot of copying from one buffer to another. The third alternative, although inelegant, could be very fast relative to the other methods, especially on a large number of processes with a small blocksize: the sort would be a very fast operation, and there would be almost no overhead associated with the exchanges other than that required for the actual message passing. However, if the blocksize is large, we'll once again run into the problem of lots of recopying of the data. In the absence of a clear choice, we'll opt for the solution we consider the most natural and elegant: derived types.

13.2.2 Details of the Hypercube Algorithm

As we mentioned earlier, it is usually easiest to understand the details of algorithms for hypercubes if we look at process ranks as binary integers rather than decimal integers. Recall that during the initial stage, two processes communicate if their addresses differ only in the most significant bit. For example, process 2, binary 010_2, exchanges with process 6, binary 110_2. So how do processes 2 and 6 determine the ranks of their partners? More generally, how do we determine the process that differs in rank from `my_rank` in the most significant bit? Conceptually, we just look at the most significant bit in `my_rank`; if it's a 0, we change it to a 1; if it's a 1, we change it to a 0. However, it's probably not immediately obvious how we program a computer to do this; at least, it's probably not obvious if you aren't accustomed to using C's bitwise operators. If you are, you'll recognize this as a clear case for a "bitwise exclusive or."

So recall the definition of "exclusive or":

A	*B*	*A eor B*
0	0	0
0	1	1
1	0	1
1	1	0

Observe that if we "exclusive or" a bit (0 or 1) with a 1, the resulting bit will be inverted ($0 \rightarrow 1$, $1 \rightarrow 0$), while if we "exclusive or" a bit with a 0, the resulting

bit will be the same as the original bit. So if we can compute an integer whose binary representation consists of a 1 in the most significant bit and 0 everywhere else, we can "bitwise exclusive or" this integer with `my_rank` to compute the rank of our partner during stage 0. If there are $p = 2^d$ processes, then the most significant bit is the dth bit, and $d = \log_2(p)$ can be computed by counting the number of times we need to right-shift the binary representation of p until the result is 1. For example, $8 = 1000_2 = 2^3$, and if we right-shift 1000_2 three times, the result will be 1. Thus, we can write a simple $\log_2$ function to get the number of bits:

```
int log_base2(int p) {
/* Just counts number of bits to right of most significant
 * bit.  So for p not a power of 2, it returns the floor
 * of log_2(p).
 */
    int return_val = 0;
    unsigned q;

    q = (unsigned) p;
    while(q != 1) {
        q = q >> 1;
        return_val++;
    }
    return return_val;
}  /* log_base2 */
```

Since some implementations of C fill leading bits with a sign bit instead of a 0 when an int is right-shifted, we copied p into a variable having type unsigned before shifting; C does guarantee that leading bits of an unsigned will be filled with 0s.

Now we can use the result of this operation to compute our `eor_bit` and the rank of our `partner`.

```
    unsigned eor_bit;
    int d;
    int partner;

    d = log_base2(p);
    eor_bit = 1 << (d-1); /* fills low-order bits with 0s */
    partner = my_rank ^ eor_bit; /* ^ = bitwise eor in C */
```

Of course, we also need to calculate our partner at subsequent stages, but recall that during the next stage, our partner's rank differs from ours in the second most significant bit. Thus, if we right-shift our `eor_bit` by 1 bit, we can compute the rank of the partner during the next stage. In general, there will be d exchanges or d stages, and the basic outline of the hypercube exchange will be something like this:

```
/* eor_bit should be unsigned so that
 * right shift will fill leftmost bits with 0
 */
d = log_base2(p);
eor_bit = 1 << (d-1);

for (stage = 0; stage < d; stage++) {
    partner = my_rank ^ eor_bit;
    Build derived data type;
    Exchange data;
    eor_bit = eor_bit >> 1;
}
```

The derived datatypes we use in the exchanges will, in general, consist of equally spaced blocks of floats. `MPI_Type_vector` is specifically designed for this purpose. Recall the syntax:

```
int MPI_Type_vector(
    int           number_of_blocks  /* in  */,
    int           blocksize         /* in  */,
    int           stride            /* in  */,
    MPI_Datatype  old_type          /* in  */,
    MPI_Datatype* new_type          /* out */)
```

The `stride` is just the number of elements between the start of successive blocks. For example, suppose we have an array x containing 16 floats, and we want to send x_0, x_1, x_8, and x_9 to another process. Then we can use a type built with `MPI_Type_vector`. The `number_of_blocks` is 2, the `blocksize` is 2, the `stride` is 8, and the `old_type` is `MPI_FLOAT`.

For our allgather function, the blocksize is in the parameter list. If we take a look at Figures 13.7–13.9, we'll see that each process sends one block during stage 0, two blocks during stage 1, and four blocks during stage 2. In general, then, we start with one block and double `number_of_blocks` at each stage. If we again look at Figures 13.7–13.9, it's not immediately clear what the `stride` should be during stage 0, since we're only sending a single block. However, during stage 1, the `stride` is four, and during stage 2, the `stride` is two. Since there is only one block exchanged during stage 0, we can make the `stride` anything we want. So in order to be consistent with the remaining stages, we can make the `stride` eight at stage 0. Note that in general, these `strides` should be multiplied by the blocksize. To summarize, during stage `stage`, we can build the derived datatype with

```
number_of_blocks = 1 << stage;       /* 2^stage     */
stride = (1 << (d-stage))*blocksize; /* 2^(d-stage) */

MPI_Type_vector(number_of_blocks, blocksize,
    stride, MPI_FLOAT, &hole_type)
```

Table 13.1 Offsets in eight-processor pairwise exchange (Pt. = partner, S/O = send offset, R/O = receive offset)

	Stage								
	0			1			2		
Process	*Pt.*	*S/O*	*R/O*	*Pt.*	*S/O*	*R/O*	*Pt.*	*S/O*	*R/O*
0 = 000	100	000	100	010	000	010	001	000	001
1 = 001	101	001	101	011	001	011	000	001	000
2 = 010	110	010	110	000	010	000	011	000	001
3 = 011	111	011	111	001	011	001	010	001	000
4 = 100	000	100	000	110	000	010	101	000	001
5 = 101	001	101	001	111	001	011	100	001	000
6 = 110	010	110	010	100	010	000	111	000	001
7 = 111	011	111	011	101	011	001	110	001	000

The only remaining issue is the offsets to be used in the sends and receives. Once again, using binary representations of the offsets and process ranks makes it easier to understand the pattern. In Table 13.1, we've summarized the offsets from our eight-process example. Let's take a look at the send offsets. Rules such as "My send offset during the current stage is the minimum of my send offset and my receive offset during the last stage" don't seem as elegant as the underlying binary pattern: start with send offset `my_rank`; during the next stage change the most significant bit to a zero; during the next stage change the next most significant bit to a zero; etc. It looks once again like a bitwise operation: leave all the bits of the previous send offset, except one, intact. For this single bit, leave it alone if it's a 0, and change it to a 0 if it's a 1. You may have already guessed that the appropriate operator is "bitwise and." Recall the table defining "bitwise and".

A	*B*	*A and B*
0	0	0
0	1	0
1	0	0
1	1	1

If we "and" a bit with a 1, the resulting bit is the same as the original bit, while if we "and" a bit with a 0, the resulting bit is 0, regardless of what the original bit was. So we would like our bitmask to consist initially of all 1s, and then we can right-shift it after each stage. To do this, we can compute $p = 2^d = 100\ldots00_2$, and subtract 1, $100\ldots00_2 - 1_2 = 011\ldots11_2$. In C, this is

```
unsigned and_bits;
int d;
int send_offset;
```

```
d = log_base2(p);
and_bits = (1 << d) - 1; /* or just p - 1 */
```

Now the send offset can be computed at each stage as follows:

```
for (stage = 0; stage < d; stage++) {
    send_offset = (my_rank & and_bits)*blocksize;
      :
    and_bits = and_bits >> 1;
} /* for stage */
```

Notice that we need to multiply the general send offset by `blocksize`.

Now it's easy to figure out what the receive offset should be! Our partner is using the same formula to calculate the send offset that we're using, except that she's using her rank. Thus

```
recv_offset = (partner & and_bits)*blocksize
```

To summarize, here's the code:

```
void Allgather_cube(
        float    x[]        /* in  */,
        int      blocksize  /* in  */,
        float    y[]        /* out */,
        MPI_Comm comm       /* in  */) {

    int           i, d, p, my_rank;
    unsigned      eor_bit;
    unsigned      and_bits;
    int           stage, partner;
    MPI_Datatype  hole_type;
    int           send_offset, recv_offset;
    MPI_Status    status;

    int log_base2(int p);

    MPI_Comm_size(comm, &p);
    MPI_Comm_rank(comm, &my_rank);

    /* Copy x into correct location in y */
    for (i = 0; i < blocksize; i++)
        y[i + my_rank*blocksize] = x[i];

    /* Set up */
    d = log_base2(p);
    eor_bit = 1 << (d-1);
    and_bits = (1 << d) - 1;
```

```
    for (stage = 0; stage < d; stage++) {
        partner = my_rank ^ eor_bit;
        send_offset = (my_rank & and_bits)*blocksize;
        recv_offset = (partner & and_bits)*blocksize;

        MPI_Type_vector(1 << stage, blocksize,
            (1 << (d-stage))*blocksize, MPI_FLOAT,
            &hole_type);
        MPI_Type_commit(&hole_type);

        MPI_Send(y + send_offset, 1, hole_type,
            partner, 0, comm);
        MPI_Recv(y + recv_offset, 1, hole_type,
            partner, 0, comm, &status);

        MPI_Type_free(&hole_type);  /* Free type so we  */
                  /* can build new type during next pass */
        eor_bit = eor_bit >> 1;
        and_bits = and_bits >> 1;
    }
} /* Allgather_cube */
```

A couple of caveats are in order. The program assumes that there will be some system buffering, since it executes all sends first. Also, this code may fail with a blocksize of 0: `hole_type` may be an invalid datatype.

13.3 Send-receive

We've remarked after completing both `Allgather_ring` and `Allgather_cube` that the functions could fail if there is no system buffering. We briefly discussed this issue in Chapter 5: in MPI parlance our functions are *unsafe.* A couple of natural questions might occur: why are they unsafe, and, if they are, what can we do to make them safe?

First, why are the functions unsafe? Consider the case that we have two processes. Then there will be a single exchange of data between process 0 and process 1. If we synchronize the processes before the exchanges we'll have approximately the following sequence of events:

Time	*Process 0*	*Process 1*
1	MPI_Send to 1	MPI_Send to 0
2	MPI_Recv from 1	MPI_Recv from 0

However, if there's no buffering, the `MPI_Sends` will never return: when a blocking function is called, the function won't return until it's safe for the program to modify the arguments to the function, and if there's no buffering,

the buffer passed to MPI_Send can't be modified until the data has been copied over to the other processor—i.e., until MPI_Recv is called. So process 0 will wait in MPI_Send until process 1 calls MPI_Recv, and process 1 will wait in MPI_Send until process 0 calls MPI_Recv. We've encountered this situation several times before: it's called *deadlock*.

Of course, the harder question is how to make them safe. We have basically two options: we can figure out how to make them safe by reorganizing the sends and receives, or we can let MPI make them safe.

In order to make them safe by reorganizing the sends and receives, we need to decide who will send first and who will receive first. A standard solution is to have the processors with even ranks send first, and the processors with odd ranks receive first. So the sequence of events in our function would now be

Time	*Process 0*	*Process 1*
1	MPI_Send to 1	MPI_Recv from 0
2	MPI_Recv from 1	MPI_Send to 0

Note that this solution works even if we have an odd number of processes. For example, if we used alternating sends and receives in our ring pass with three processes, the approximate sequence of events might be as follows:

Time	*Process 0*	*Process 1*	*Process 2*
1	MPI_Send to 1	MPI_Recv from 0	Start MPI_Send to 0
2	MPI_Recv from 2	Start MPI_Send to 2	Finish MPI_Send
3	Done	Finish MPI_Send	MPI_Recv from 1

So there may be a delay in the completion of one communication, but the program will complete successfully.

A simpler solution is to let MPI take care of the problem. In fact, we encountered this solution in our implementation of Fox's algorithm (Chapter 7). The function MPI_Sendrecv, as its name implies, performs both a send and a receive, and it organizes them so that even in systems with no buffering the calling program won't deadlock—at least not in the way that the MPI_Send/MPI_Recv implementation deadlocks! The syntax of MPI_Sendrecv is

```
int MPI_Sendrecv(
        void*        send_buf    /* in  */,
        int          send_count  /* in  */,
        MPI_Datatype send_type   /* in  */,
        int          destination /* in  */,
        int          send_tag    /* in  */,
        void*        recv_buf    /* out */,
        int          recv_count  /* in  */,
        MPI_Datatype recv_type   /* in  */,
        int          source      /* in  */,
```

```
        int          recv_tag    /* in  */,
        MPI_Comm     comm        /* in  */,
        MPI_Status*  status      /* out */ )
```

Notice that the parameter list is basically just a concatenation of the parameter lists for `MPI_Send` and `MPI_Recv`. The only difference is that the communicator parameter is not repeated. The `destination` and the `source` parameters can be the same. The "send" in an `MPI_Sendrecv` can be matched by an ordinary `MPI_Recv`, and the "receive" can be matched by an ordinary `MPI_Send`. The basic difference between a call to this function and `MPI_Send` followed by `MPI_Recv` (or vice versa) is that MPI can try to arrange that no deadlock occurs since it knows that the sends and receives will be paired. We leave it as an exercise to modify `Allgather_ring` and `Allgather_cube` so that they use the `MPI_Sendrecv`.

Recollect that MPI doesn't allow a single variable to be passed to two distinct parameters if one of them is an output parameter. Thus, we can't call `MPI_Sendrecv` with `send_buf = recv_buf`. Since it is very common in practice for paired send/receives to use the same buffer, MPI provides a variant that does allow us to use a single buffer:

```
int MPI_Sendrecv_replace(
        void*        buffer      /* in/out */,
        int          count       /* in     */,
        MPI_Datatype datatype    /* in     */,
        int          destination /* in     */,
        int          send_tag    /* in     */,
        int          recv_tag    /* in     */,
        MPI_Comm     comm        /* in     */,
        MPI_Status*  status      /* out    */)
```

Note that this implies the existence of some system buffering.

13.4 Null Processes

A typical use of `MPI_Sendrecv` and `MPI_Sendrecv_replace` is to shift data in a process ring: this is exactly what we do in `Allgather_ring`. How would we manage this if, instead of a process ring, we had a linear array of processes? The obvious answer is we would have to test for boundary processes:

```
if (my_rank == 0)
   MPI_Send(send_buf, count, . . . );
else if (my_rank == p-1)
   MPI_Recv(recv_buf, count, . . . );
else
   MPI_Sendrecv(send_buf, count, . . . );
```

An elegant alternative is provided by MPI in the dummy process MPI_PROC_NULL. If a process executes a send with destination MPI_PROC_NULL, the send simply returns as soon as possible, with no change to its arguments. If a process executes a receive with source MPI_PROC_NULL, the receive returns without modifying the receive buffer. The return value of status.MPI_Source is MPI_PROC_NULL, and the return value of status.MPI_Tag is the MPI constant MPI_ANY_TAG. If we call

```
MPI_Get_count(&status, datatype, &count),
```

the value returned in count will be 0.

In our example, if we set source = MPI_PROC_NULL on process 0 and destination = MPI_PROC_NULL on process $p - 1$, then each process can call MPI_Sendrecv.

Another example of the use of MPI_PROC_NULL is provided by Allgather_cube. It can be modified so that p can be any positive integer, not just a power of two. If we test the value of partner before the communication and find that its value is greater than $p - 1$, we can assign it the value MPI_PROC_NULL. We'll discuss the details of this modification in a programming assignment.

13.5 Nonblocking Communication

Since MPI_Send, MPI_Recv, and MPI_Sendrecv are blocking operations, they will not return until the arguments to the functions can be safely modified by subsequent statements in the program. For MPI_Send, this means that the message envelope has been created and the message has been sent or that the contents of the message have been copied into a system buffer. For MPI_Recv, it means that the message has been received into the buffer specified by the buffer argument. For MPI_Sendrecv, it implies that the outgoing message has been sent or buffered, and the incoming message has been received.

All of these semantics may imply that the resources available to the sending or receiving process are not being fully utilized. For example, if the MPI process controls two physical processors, a processor for computation and a processor for communication, then the send operation should be able to proceed concurrently with some computation, as long as the computation doesn't modify any of the arguments to the send operation. Also if, during execution of a receive operation, a process finds that the data to be received is not yet available, then the process should be able to continue with useful computation as long as it doesn't interfere with the arguments to the receive.

Nonblocking communication is explicitly designed to meet these needs. A call to a nonblocking send or receive simply starts, or **posts**, the communication operation. It is then up to the user program to explicitly complete the communication at some later point in the program. Thus, any nonblocking op-

eration requires a minimum of two function calls: a call to start the operation and a call to complete the operation.

The basic functions in MPI for starting nonblocking communications are `MPI_Isend` and `MPI_Irecv`. The "I" stands for "immediate," i.e., they return (more or less) immediately. Their syntax is very similar to the syntax of `MPI_Send` and `MPI_Recv`:

```
int MPI_Isend(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)
```

and

```
int MPI_Irecv(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          source      /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)
```

The parameters that they have in common with `MPI_Send` and `MPI_Recv` have the same meaning. However, the semantics are different. Both calls only *start* the operation. For `MPI_Isend` this means that the system has been informed that it can start copying data out of the send buffer (either to a system buffer or to the destination). For `MPI_Irecv`, it means that the system has been informed that it can start copying data into the buffer. Neither send nor receive buffers should be modified until the operations are explicitly completed or cancelled.

The `request` parameter is a *handle* associated to an *opaque object*. Recollect that this means, effectively, that the object referenced by `request` is system defined, and that it cannot be directly accessed by the user. Its purpose is to identify the operation started by the nonblocking call. So it will contain information on such things as the source or destination, the tag, the communicator, and the buffer. When the nonblocking operation is completed, the `request` initialized by the call to `MPI_Isend` or `MPI_Irecv` is used to identify the operation to be completed.

There are a variety of functions that MPI uses to complete nonblocking operations. The simplest of these is `MPI_Wait`. It can be used to complete any nonblocking operation.

```
int MPI_Wait(
    MPI_Request* request  /* in/out */,
    MPI_Status*  status   /* out    */ )
```

The request parameter corresponds to the request parameter returned by MPI_Isend or MPI_Irecv. MPI_Wait blocks until the operation identified by request completes: if it was a send, either the message has been sent or buffered by the system; if it was a receive, the message has been copied into the receive buffer. When MPI_Wait returns, request is set to MPI_REQUEST_NULL. This means that there is no pending operation associated to request. If the call to MPI_Wait is used to complete an operation started by MPI_Irecv, the information returned in the status parameter is the same as the information returned in status by a call to MPI_Recv.

Finally, it should be noted that it is perfectly legal to match blocking operations with nonblocking operations. For example, a message sent with MPI_Isend can be received by a call to MPI_Recv.

13.5.1 Ring Allgather with Nonblocking Communication

Let's illustrate the use of nonblocking communication with our allgather functions.

When we use nonblocking communication, we want to do as much *local* computation as possible between the start of the nonblocking operation and the call to MPI_Wait. Thus, when we convert code that uses blocking communications to the use of nonblocking communications, we usually reorganize the statements so that we can defer the completion calls. The basic loop in the blocking version of Allgather_ring looks like this:

```
for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    recv_offset = ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Send(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
        predecessor, 0, ring_comm, &status);
}
```

In order to somewhat overlap communication and computation, we can compute the offsets for the next pass through the loop before completing the communications. This will necessitate our initializing the offsets before we enter the loop:

```
MPI_Request send_request;
MPI_Request recv_request;
      :
send_offset = my_rank*blocksize;
```

Table 13.2 Runtimes of `Allgather_ring` (times are in milliseconds; blocksize = 1; both systems running `mpich`)

	Paragon		*SP2*	
Processes	*Blocking*	*Nonblocking*	*Blocking*	*Nonblocking*
2	0.14	0.10	0.11	0.080
8	1.0	0.94	0.85	0.54
32	4.9	4.2	3.9	2.5

```
recv_offset = ((my_rank - 1 + p) % p)*blocksize;
for (i = 0; i < p - 1; i++) {
    MPI_Isend(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm, &send_request);
    MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT,
        predecessor, 0, ring_comm, &recv_request );

    send_offset = ((my_rank - i - 1 + p) % p)*blocksize;
    recv_offset = ((my_rank - i - 2 + p) % p)*blocksize;

    MPI_Wait(&send_request, &status);
    MPI_Wait(&recv_request, &status);
}
```

Table 13.2 shows the runtimes of both the blocking and nonblocking versions of `Allgather_ring` on a Paragon and an SP2. If we take into consideration the relatively small amount of computation between the calls to `MPI_Isend` and `MPI_Irecv` and the calls to `MPI_Wait`, the performance improvements, especially on the SP2, are quite good.

13.5.2 Hypercube Allgather with Nonblocking Communication

Recollect that during each pass through the main loop of `Allgather_cube`, we build a new derived datatype. Since this is a fairly expensive operation, it would seem that we might obtain a significant increase in performance if we can overlap the building of the datatype with the communications. So recall the blocking code:

```
for (stage = 0; stage < d; stage++) {
    partner = my_rank ^ eor_bit;
    send_offset = (my_rank & and_bits)*blocksize;
    recv_offset = (partner & and_bits)*blocksize;

    MPI_Type_vector(1 << stage, blocksize,
        (1 << (d-stage))*blocksize, MPI_FLOAT,
        &hole_type);
    MPI_Type_commit(&hole_type);
```

```
        MPI_Send(y + send_offset, 1, hole_type,
            partner, 0, comm);
        MPI_Recv(y + recv_offset, 1, hole_type,
            partner, 0, comm, &status);

        MPI_Type_free(&hole_type);  /* Free type so we  */
                /* can build new type during next pass */
        eor_bit = eor_bit >> 1;
        and_bits = and_bits >> 1;
    }
```

It seems that we can indeed make the overlap, but note that once again, we will have to perform some extra initialization.

```
    partner = my_rank ^ eor_bit;
    send_offset = (my_rank & and_bits)*blocksize;
    recv_offset = (partner & and_bits)*blocksize;
    MPI_Type_contiguous(blocksize, MPI_FLOAT, &hole_type);
    MPI_Type_commit(&hole_type);

    for (stage = 0; stage < d; stage++) {
        MPI_Isend(y + send_offset, 1, hole_type,
            partner, 0, comm, &send_request);
        MPI_Irecv(y + recv_offset, 1, hole_type,
            partner, 0, comm, &recv_request);

        if (stage < d-1) {
            eor_bit >>= 1;
            and_bits >>= 1;
            partner = my_rank ^ eor_bit;
            send_offset = (my_rank & and_bits)*blocksize;
            recv_offset = (partner & and_bits)*blocksize;
            MPI_Type_free(&hole_type);
            MPI_Type_vector(1 << (stage+1), blocksize,
                (1 << (d-stage-1))*blocksize, MPI_FLOAT,
                &hole_type);
            MPI_Type_commit(&hole_type);
        }

        MPI_Wait(&send_request, &status);
        MPI_Wait(&recv_request, &status);
    }  /* for */
```

Note that since the first derived datatype consists of a single block, we can build it during the initialization phase using `MPI_Type_contiguous` instead of `MPI_Type_vector`.

Table 13.3 shows the runtimes of both the blocking and nonblocking versions of `Allgather_cube` on a Paragon and an SP2. Once again the perfor-

Table 13.3 Runtimes of `Allgather_cube` (times are in milliseconds; blocksize = 16,384; both systems running `mpich`)

	Paragon		*SP2*	
Processes	*Blocking*	*Nonblocking*	*Blocking*	*Nonblocking*
2	11	11	12	12
8	58	51	49	45
32	270	250	220	190

mance improvements are good, although not as substantial as they were for `Allgather_ring`.

13.6 Persistent Communication Requests

It is not uncommon for message-passing programs to repeatedly call communication functions inside a loop with exactly the same arguments. For example, consider the calls to `MPI_Isend` and `MPI_Irecv` in our ring allgather:

```
MPI_Isend(y + send_offset, blocksize, MPI_FLOAT,
    successor, 0, ring_comm, &send_request);
MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT,
    predecessor, 0, ring_comm, &recv_request );
```

Observe that during each pass, the only argument that changes in each call is the buffer argument: `send_offset` and `recv_offset` are recomputed during each pass, but the other arguments are identical. If we pack the contents of `y + send_offset` into a buffer and send the buffer, and then receive into another buffer and unpack its contents into `y + recv_offset`, then all our calls to `MPI_Isend` will use exactly the same arguments and all our calls to `MPI_Irecv` will use exactly the same arguments.

Clearly, the repeated calls to, say, `MPI_Isend` are performing redundant work. For example, the envelope of all the messages will contain exactly the same information. Thus, we might get some further improvement in performance if we could get MPI to do this redundant work just once, and, as you probably anticipated, we can. The mechanism is called a **persistent communication request.** The idea is that we do all the work (building envelopes, etc.) associated with the message passing just once; i.e., we build the appropriate request just once. Then we can reuse the request for all the appropriate communications, rather than continually rebuilding them.

Here's how this is done if we pack/unpack the data in our ring allgather:

```
#define MAX_BYTES MAX*sizeof(float)

float send_buf[MAX];
```

```
float recv_buf[MAX];
int position;
      :
MPI_Send_init(send_buf, blocksize*sizeof(float),
    MPI_PACKED, successor, 0, ring_comm,
    &send_request);
MPI_Recv_init(recv_buf, blocksize*sizeof(float),
    MPI_PACKED, predecessor, 0, ring_comm,
    &recv_request );

send_offset = my_rank*blocksize;
for (i = 0; i < p - 1; i++) {
    position = 0;
    MPI_Pack(y+send_offset, blocksize, MPI_FLOAT,
        send_buf, MAX_BYTES, &position, ring_comm);
    MPI_Start(&send_request);
    MPI_Start(&recv_request);
    recv_offset = send_offset =
       ((my_rank - i - 1 + p) % p)*blocksize;
    position = 0;
    MPI_Wait(&send_request, &status);
    MPI_Wait(&recv_request, &status);
    MPI_Unpack(recv_buf, MAX_BYTES, &position,
        y+recv_offset, blocksize, MPI_FLOAT, ring_comm);
}
MPI_Request_free(&send_request);
MPI_Request_free(&recv_request);
```

The calls to MPI_Pack and MPI_Unpack make this code appear radically different from the earlier nonblocking version. However, if you put these calls aside, it becomes apparent that there are three main differences. The first is the calls to MPI_Send_init and MPI_Recv_init before the loop: these "allocate" the persistent requests. Then, in order to actually start the communications, there are two calls to MPI_Start. Finally, since the requests are "persistent," it is up to us to free them. We do this with the calls to MPI_Request_free. We'll discuss each of these new functions in turn.

The parameter lists for the calls to MPI_Send_init and MPI_Recv_init are identical to those for MPI_Isend and MPI_Irecv:

```
int MPI_Send_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)
```

and

```
int MPI_Recv_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          source      /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)
```

However, their effect is simply to set up persistent requests for a send and a receive, respectively.

The calls to MPI_Start actually start the communication. The syntax is simply

```
int MPI_Start(
        MPI_Request* request /* in/out */ )
```

Thus, the pair of calls

```
MPI_Send_init( . . . , &persistent_send_request);
MPI_Start(&persistent_send_request);
```

has much the same effect as the single call

```
MPI_Isend( . . . , &send_request);
```

Similarly the pair of calls

```
MPI_Recv_init(. . . , &persistent_recv_request);
MPI_Start(&persistent_recv_request);
```

has much the same effect as the single call

```
MPI_Irecv( . . . , &recv_request);
```

The key difference lies in the way MPI_Wait deals with the requests. In the case of the persistent requests, MPI_Wait does not set the requests to MPI_REQUEST_NULL after completion. In the parlance of MPI, the persistent requests become **inactive.** This means simply that there is no pending operation associated to the request (e.g., a send or receive), and the requests can be reactivated by calls to MPI_Start. In order to deallocate a request, we call

```
MPI_Request_free(&request)
```

This has the effect of setting request to MPI_REQUEST_NULL.

It should be noted that sends or receives started with MPI_Start can be matched by any other type of receive or send, respectively.

Table 13.4 Runtimes of `Allgather_ring` (times are in milliseconds; blocksize = 1; B = blocking, NB = nonblocking, P = persistent; both systems running `mpich`)

	Paragon			*SP2*		
Processes	*B*	*NB*	*P*	*B*	*NB*	*P*
2	0.14	0.10	0.21	0.11	0.08	0.09
8	1.0	0.94	1.1	0.85	0.54	0.53
32	4.9	4.2	4.4	3.9	2.5	2.6

Table 13.4 shows the runtimes of all three versions of `Allgather_ring` on a Paragon and an SP2. Generally the times for the version that uses persistent requests are comparable to the times for the version that uses basic nonblocking communication. The cost of the calls to `MPI_Pack` and `MPI_Unpack` offsets the savings obtained by not recreating the requests. Indeed in some cases (Paragon, $p = 2, 8$), the cost of these calls makes the version that uses persistent requests slower than the version that uses blocking communication.

13.7 Communication Modes

At several points during our discussions, in both this and other chapters, we've encountered the idea of *safety* in MPI. Recollect that a program is safe if it will produce correct results *even if the system provides no buffering.* Most programmers of message-passing systems expect the system to provide some buffering, and, as a consequence, they routinely write unsafe programs. Consider, for example, our initial implementations of `Allgather_ring` and `Allgather_cube`.

If we are writing a program that must absolutely be portable to any system, we can guarantee the safety of our program in two ways:

1. We can reorganize our communications so that the program will not deadlock if sends cannot complete until a matching receive is posted. We discussed an example of this in section 13.3.
2. A possibly less painful solution is to organize our own buffering.

In either case, we are, effectively, changing the **communication mode** of our program.

There are four communication modes in MPI: **standard, buffered, synchronous,** and **ready.** They correspond to four different types of send operations. There is only a standard mode for receive operations. In standard mode, it is up to the system to decide whether messages should be buffered. A typical implementation might buffer only relatively small messages. Up until now all of our sends, whether blocking, nonblocking, or persistent, have used the standard mode.

13.7.1 Synchronous Mode

In synchronous mode a send won't complete until a matching receive has been posted and the matching receive has begun reception of the data. MPI provides three synchronous mode send operations:

```
int MPI_Ssend(
    void*        buffer      /* in */,
    int          count       /* in */,
    MPI_Datatype datatype    /* in */,
    int          destination /* in */,
    int          tag         /* in */,
    MPI_Comm     comm        /* in */ );

int MPI_Issend(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */ );

int MPI_Ssend_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */ );
```

Their effect is much the same as the corresponding standard mode sends. However, `MPI_Ssend` and the waits corresponding to `MPI_Issend` and `MPI_Ssend_init` will not complete until the corresponding receives have started. Thus, synchronous mode sends require no system buffering, and we can assure that our program is safe if it runs correctly using only synchronous mode sends.

Observe that if we replace the standard sends in, say, `Allgather_ring` with synchronous sends, then the function will hang, since none of the sends will complete until the corresponding receive has started. Thus, if we wish to use synchronous sends, we need to do something like this:

```
if ((my_rank % 2) == 0){ /* Even ranks send first */
    MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
       predecessor, 0, ring_comm, &status);
```

```
} else {  /* Odd ranks receive first */
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
       predecessor, 0, ring_comm, &status);
    MPI_Ssend(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm);
}
```

13.7.2 Ready Mode

On some systems it's possible to improve the performance of a message transmission if the system knows, before a send has been initiated, that the corresponding receive has already been posted. For such systems, MPI provides the ready mode. The parameter lists of the ready sends are identical to the parameter lists for the corresponding standard sends:

```
int MPI_Rsend(
    void*        buffer      /* in */,
    int          count       /* in */,
    MPI_Datatype datatype    /* in */,
    int          destination /* in */,
    int          tag         /* in */,
    MPI_Comm     comm        /* in */ );

int MPI_Irsend(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */ );

int MPI_Rsend_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */ );
```

The only difference between the semantics of the various ready sends and the corresponding standard sends is that the ready sends are erroneous if the matching receive hasn't been posted, and the behavior of an erroneous program is unspecified. Also note that it should be possible to replace all ready sends in a program with standard sends without affecting the output, although the program with the standard sends may be slower.

If we don't have too many processes, we could rewrite our allgathers by posting all the receives with `MPI_Irecv` and then executing the sends with one of the ready sends. For example, we might do something like this in `Allgather_ring`:

```
MPI_Request  request[p-1];

for (i = 0; i < p - 1; i++) {
    recv_offset =
        ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Irecv(y + recv_offset, blocksize, MPI_FLOAT,
        predecessor, i, ring_comm, &(request[i]));
}

MPI_Barrier(ring_comm);

for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    MPI_Rsend(y + send_offset, blocksize, MPI_FLOAT,
        successor, i, ring_comm);
    MPI_Wait(&(request[i]), &status);
}
```

Note that the call to `MPI_Barrier` is necessary since otherwise some process might enter the loop of sends before a receive had been posted.

13.7.3 Buffered Mode

The final mode, and the only one that has additional associated functions, is buffered mode. In buffered mode, a send operation is **local**. In other words, its completion does not depend on the existence of a matching receive. The other send modes are **nonlocal**; i.e., their completion may depend on the existence of a matching receive posted by another process. If a send is started in buffered mode and the matching receive hasn't been posted, the process *must* buffer the data. Of course, it's possible to exceed the available buffer space, in which case the program is erroneous.

The buffered mode sends have the same parameter lists as the other sends.

```
int MPI_Bsend(
    void*        buffer      /* in */,
    int          count       /* in */,
    MPI_Datatype datatype    /* in */,
    int          destination /* in */,
    int          tag         /* in */,
    MPI_Comm     comm        /* in */ );

int MPI_Ibsend(
    void*        buffer      /* in  */,
```

```
    int           count       /* in  */,
    MPI_Datatype  datatype    /* in  */,
    int           destination /* in  */,
    int           tag         /* in  */,
    MPI_Comm      comm        /* in  */,
    MPI_Request*  request     /* out */ );

int MPI_Bsend_init(
    void*         buffer      /* in  */,
    int           count       /* in  */,
    MPI_Datatype  datatype    /* in  */,
    int           destination /* in  */,
    int           tag         /* in  */,
    MPI_Comm      comm        /* in  */,
    MPI_Request*  request     /* out */ );
```

The semantics are much the same as the corresponding standard sends. However, as we just noted, the system *must* buffer the data and return if the matching receive hasn't already been posted.

The missing ingredient in buffered mode is the buffer. In order to use buffered mode, it's up to the user, not the system, to allocate the buffer. This can be done with `MPI_Buffer_attach`:

```
int MPI_Buffer_attach(
        void* buffer      /* in */,
        int   buffer_size /* in */)
```

This is used by the system to buffer messages sent in buffered mode only, and there can only be one buffer attached at any time.

Thus, if we wanted to guarantee the safety of our program, we could try to estimate the maximum amount of data that would be buffered at any given time, attach a buffer of this size, and carry out all our sends in buffered mode.

Finally, if we know that we no longer need a buffer, it can be released from control by MPI with

```
int MPI_Buffer_detach(
        void* buffer_address /* out */,
        int*  size_ptr       /* out */ )
```

This function removes the previously attached buffer from control by the system. If there are pending buffered sends, it will block until they have completed. Thus, when it completes, the user program can reuse or free the buffer. Note that it returns a pointer to the previously attached buffer and a pointer to its size. (Recollect that `void` indicates a pointer to an arbitrary type, including a pointer to a pointer.)

Thus, we could guarantee the safety of, say, `Allgather_ring` by just preceding the loop of sends and receives with a call to `MPI_Buffer_attach`, and following the loop with a call to `MPI_Buffer_free`:

```
char buffer[MAX_BUF];
int  buffer_size = MAX_BUF;
      :
MPI_Buffer_attach(buffer, buffer_size);

for (i = 0; i < p - 1; i++) {
    send_offset = ((my_rank - i + p) % p)*blocksize;
    recv_offset =
        ((my_rank - i - 1 + p) % p)*blocksize;
    MPI_Bsend(y + send_offset, blocksize, MPI_FLOAT,
        successor, 0, ring_comm);
    MPI_Recv(y + recv_offset, blocksize, MPI_FLOAT,
        predecessor, 0, ring_comm, &status);
}

MPI_Buffer_detach(&buffer, &buffer_size);
```

Be aware that this may not work if some other part of the program has already attached a buffer. Thus, we should only use buffered mode sends if we know whether other parts of the program are using them.

13.8 The Last Word on Point-to-Point Communication

As you've probably already observed, MPI provides a very rich set of point-to-point communication functions, and there are more than a few that we haven't examined in this chapter. We will explore more point-to-point functions in the exercises. However, for the sake of completeness, we'll list a couple of the categories of functions we have not yet covered.

Most of the remaining functions have to do with completing nonblocking operations. There are variants of `MPI_Wait` that have arrays of requests as parameters. These variants can be used to wait on any one of the requests, a subset of the requests, or all of the requests.

There is also a function, `MPI_Test`, that can be used simply to check whether an operation has completed. If the operation hasn't completed, the program can carry out further local computation, while if it has, `MPI_Test` will behave in the same manner as `MPI_Wait`. There are also variants of `MPI_Test` that can be used to check for the completion of multiple operations.

13.9 Summary

We've covered quite a lot of material in this chapter. We studied (in gruesome detail) two algorithms for allgather and in the process learned quite a bit about programming a hypercube and using bit operations in C.

After coding the two algorithms, we discussed `MPI_Sendrecv` and `MPI_Sendrecv_replace`:

```
int MPI_Sendrecv(
        void*        send_buf    /* in  */,
        int          send_count  /* in  */,
        MPI_Datatype send_type   /* in  */,
        int          destination /* in  */,
        int          send_tag    /* in  */,
        void*        recv_buf    /* out */,
        int          recv_count  /* in  */,
        MPI_Datatype recv_type   /* in  */,
        int          source      /* in  */,
        int          recv_tag    /* in  */,
        MPI_Comm     comm        /* in  */,
        MPI_Status*  status      /* out */ )

int MPI_Sendrecv_replace(
        void*        buffer      /* in/out */,
        int          count       /* in     */,
        MPI_Datatype datatype    /* in     */,
        int          destination /* in     */,
        int          send_tag    /* in     */,
        int          recv_tag    /* in     */,
        MPI_Comm     comm        /* in     */,
        MPI_Status*  status      /* out    */)
```

As their names suggest, these functions perform both a send and a receive. They also take care of any buffering of messages that may be necessary. They are especially convenient if processes are exchanging data or if there is a shift of data (e.g., 0 sends to 1, 1 sends to 2, 2 sends to 3, etc.). The second version can be used if we want to use the same storage for both the data sent and the data received.

In many cases, it's convenient to be able to send or receive data from a nonexistent process. For example, in the shift of the data in the previous paragraph, it might be useful to have process 0 receive from a nonexistent process, and process $p - 1$ send to a nonexistent process. MPI provides MPI_PROC_NULL for this purpose. Sends to and receives from MPI_PROC_NULL simply return without changing the arguments.

Perhaps the most important idea in the chapter is *nonblocking communication*. Nonblocking communication functions provide us with an explicit means for overlapping communication and computation. By using nonblocking communication we may be able to hide some of the cost of communication and obtain significantly improved performance. The basic functions for nonblocking communication are

```
int MPI_Isend(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
```

```
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)

int MPI_Irecv(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          source      /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)

int MPI_Wait(
    MPI_Request* request     /* in/out */,
    MPI_Status*  status      /* out    */ )
```

`MPI_Isend` and `MPI_Irecv` initiate a send and a receive, respectively. When `MPI_Isend` returns, however, the send may not have completed. So the contents of `buffer` should not be modified. Similarly, when `MPI_Irecv` returns, the data may not have been received and `buffer` should not be used to store other data. In order to complete either operation, we can call `MPI_Wait` with the `request` argument returned by the call to `MPI_Isend` or `MPI_Irecv`. When `MPI_Wait` returns, the send or receive will have completed. So in order to exploit the full power of nonblocking communication, we should try to structure our code so that we do useful work between a call to `MPI_Isend` or `MPI_Irecv` and the corresponding call to `MPI_Wait`.

If we are repeatedly calling `MPI_Isend` with exactly the same arguments, we will be repeatedly carrying out many of the same operations on the same data. For example, since the arguments are the same each time, the message envelope will be the same each time, but each time we call `MPI_Isend` the contents of the envelope will be recomputed. MPI provides a way for us to avoid some of this repeated effort with *persistent communication requests*. We use one function, `MPI_Send_init`, to do all the work necessary for setting up the communication only once. A call to this function will create a *persistent* or reusable request. Each time we're ready to initiate a send (with the same arguments), we call a second function, `MPI_Start`, which is passed the persistent request. When we're ready to complete the send, we can call `MPI_Wait` as usual. The same ideas can be used for receives. The syntax of the functions is

```
int MPI_Send_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          destination /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)
```

```
int MPI_Recv_init(
    void*        buffer      /* in  */,
    int          count       /* in  */,
    MPI_Datatype datatype    /* in  */,
    int          source      /* in  */,
    int          tag         /* in  */,
    MPI_Comm     comm        /* in  */,
    MPI_Request* request     /* out */)

int MPI_Start(
        MPI_Request* request /* in  */)
```

We also discussed MPI's different communication modes: standard, synchronous, ready, and buffered. All receives in MPI use standard mode; only sends can use different modes. Up to this point we've been using standard mode; in standard mode it is up to the system to decide whether a message should be buffered. In synchronous mode, a send won't complete until the matching receive has begun receiving the data, so no buffering is needed. Some systems can optimize the performance of a send if it is known that the corresponding receive has been posted before the send is initiated; in ready mode, a send is erroneous unless the corresponding receive has already been posted. We can use buffered mode to guarantee the safety of our communications. If a matching receive hasn't already been posted and the send uses buffered mode, the system must buffer the data being sent in a user-supplied buffer.

For each of the different modes, there is a blocking send, a nonblocking send, and a persistent send. They are distinguished by the addition of the letter "S" for synchronous, "R" for ready, and "B" for buffered. Thus, MPI defines

```
MPI_Ssend,       MPI_Rsend,       MPI_Bsend,
MPI_Issend,      MPI_Irsend,      MPI_Ibsend,
MPI_Ssend_init,  MPI_Rsend_init,  MPI_Bsend_init.
```

Their parameter lists are the same as the parameter lists for the corresponding standard mode sends.

In order to use the buffered mode sends the user must set aside a buffer. This can be done by calling

```
int MPI_Buffer_attach(
        void* buffer      /* in */,
        int   buffer_size /* in */)
```

The buffer size is in bytes. A buffer can be freed with

```
int MPI_Buffer_detach(
        void* buffer_address /* out */,
        int*  size_ptr       /* out */ )
```

At any time during the execution of a program there may be only one buffer attached on a process.

We closed the chapter with a brief mention of some other point-to-point communication functions. These fell into two categories: variants of `MPI_Wait`, which can be used to complete multiple nonblocking operations, and `MPI_Test`, which can be used to test for the completion of a nonblocking operation.

13.10 References

Additional information on point-to-point communication can be found in the MPI Standard [28, 29], Gropp et al. [21], and Snir et al. [34].

Kumar et al. [26] have extensive discussions of algorithms for allgather and other collective communication functions.

13.11 Exercises

1. Complete the coding of `Allgather_ring` using synchronous sends and receives. Compare its performance to the performance of the function using blocking sends and receives.
2. Complete the coding of `Allgather_ring` using ready sends. Compare its performance to the performance of the function using blocking sends and receives.
3. `MPI_Test` can be used to check whether a nonblocking operation has completed. Its syntax is

```
int MPI_Test(
        MPI_Request*  request  /* in/out */,
        int*          flag     /* out    */,
        MPI_Status*   status   /* out    */)
```

   It checks for completion of the communication operation associated with `request`. If the operation has completed, it returns a nonzero value in `flag`. Otherwise it returns 0 in `flag`. If the operation associated with the request is a receive and the operation completed, information on the receive will be returned in `status`. If the operation completed and the request was not started by a persistent operation, it will be set to `MPI_REQUEST_NULL`. If the operation completed and the request was started by a persistent operation, it will be made inactive.

   Modify `Allgather_ring` so that it starts all of its receives using nonblocking receives (be careful to use different tags!) and then checks its list of requests using `MPI_Test`. When it finds a request that has been completed, it should send the received data on to the next process. Can this function deadlock?

4. MPI_Waitany can be used to wait for one of a list of operations. Its syntax is

```
int MPI_Waitany(
        int           count        /* in     */,
        MPI_Request   requests[]   /* in/out */,
        int*          index        /* out    */,
        MPI_Status*   status       /* out    */)
```

MPI_Waitany blocks until the operation associated with one of the elements of requests has completed. The requests array consists of count elements. When the operation completes, it returns the subscript of the completed operation in *index. The element of requests is either set to MPI_REQUEST_NULL or made inactive. If the operation was a receive, status information is returned in status.

Modify the solution to exercise 3 so that it uses MPI_Waitany. How does its performance compare to the other ring allgathers?

13.12 Programming Assignments

1. Modify Allgather_cube so that it can deal with p not a power of two. The main idea is to test the value of partner: if it's greater than p, set it to MPI_PROC_NULL. However there are a number of other issues that need to be addressed. The log_base2 function needs to be changed so that it returns $\lceil p \rceil$ rather than $\lfloor p \rfloor$. A more complex problem arises from the ordering of the exchanges. For example, if $p = 8$, process 5 is first paired with process 1, then process 7, and finally process 4. If $p = 6$ and we try to use the same sequence of pairings, the data stored on processes 2 and 3 will never get to process 5. (Work an example!) The problem here is that we're moving from high order to low order; i.e., eor_bit is being *right-shifted.* If we reverse this process, we won't run into this problem. Of course, this will mean that we need to change the sequence of offsets as well.
2. Write a ring-based all-to-all function. What type of nonblocking communications should you use?
3. Write a hypercube exchange all-to-all function. What type of nonblocking communications will you use?
4. Modify your cellular automaton program (see programming assignment 4 from Chapter 10) so that it uses nonblocking sends and receives. Should your code be reorganized so that you can obtain a better overlap between communication and computation? Is there any advantage to using persistent sends and receives in your program? You may want to suppress printing of the output to study the program's performance.