3 Collective Communication

There are probably a few things in the trapezoid rule program that we can improve on. For example, there is the I/O issue. There are also a couple of problems we haven't discussed yet. Let's look at what happens when the program is run with eight processes.

All the processes begin executing the program (more or less) simultaneously. However, after carrying out the basic set-up tasks (calls to MPI_Init, MPI_Comm_size, and MPI_Comm_rank), processes 1–7 are idle while process 0 collects the input data. We don't want to have idle processes, but in view of our restrictions on which processes can read input, there isn't much we can do about this. However, after process 0 has collected the input data, the higher rank processes must continue to wait while 0 sends the input data to the lower rank processes. This isn't just an I/O issue. Notice that there is a similar inefficiency at the end of the program, when process 0 does all the work of collecting and adding the local integrals.

Of course, this is highly undesirable: the main point of parallel processing is to get multiple processes to collaborate on solving a problem. If one of the processes is doing most of the work, we might as well use a conventional, single-processor machine.

3.1 Tree-Structured Communication

Let's try to improve our code. We'll begin by focussing on the distribution of the input data. How can we divide the work more evenly among the processes? A natural solution is to imagine that we have a tree of processes, with 0 at the root.

During the first stage of the data distribution, 0 sends the data to (say) 4. During the next stage, 0 sends the data to 2, while 4 sends it to 6. During the last stage, 0 sends to 1, while 2 sends to 3, 4 sends to 5, and 6 sends to 7. (See figure 3.1.) So we have reduced our input distribution loop from 7 stages to 3 stages. More generally, if we have p processes, this procedure allows us to distribute the input data in $\lceil \log_2(p) \rceil$* stages, rather than $p - 1$ stages, which, if p is large, is a huge savings.

In order to modify the Get_data function to use a tree-structured distri-

*The notation $\lceil x \rceil$ denotes the smallest whole number greater than or equal to x.

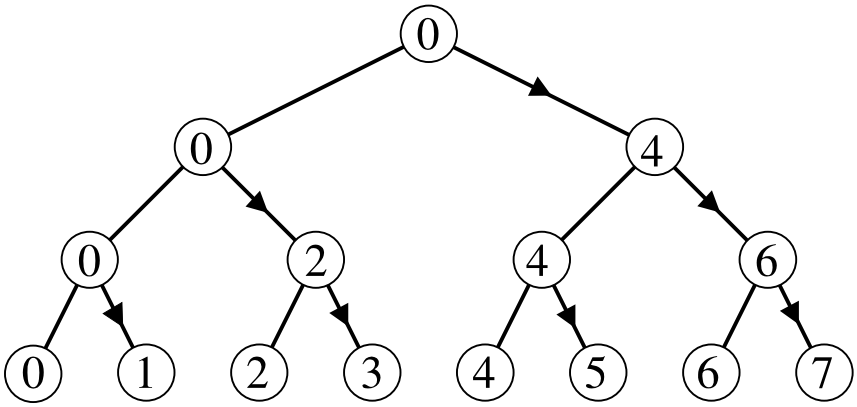

Figure 1: Processors configured as a tree

bution scheme, we need to introduce a loop with $\lceil \log_2(p) \rceil$ stages. In order to implement the loop, each process needs to calculate at each stage

- whether it receives, and, if so, the source; and
- whether it sends, and, if so, the destination.

As you can probably guess, these calculations can be a bit complicated, especially since there is no canonical choice of ordering. In our example, we chose:

1. 0 sends to 4.
2. 0 sends to 2, 4 sends to 6.
3. 0 sends to 1, 2 sends to 3, 4 sends to 5, 6 sends to 7.

We might also have chosen (for example):

1. 0 sends to 1.
2. 0 sends to 2, 1 sends to 3.
3. 0 sends to 4, 1 sends to 5, 2 sends to 6, 3 sends to 7.

Indeed, unless we know something about the underlying topology of our machine, we can't really decide which scheme is better.

So ideally we would prefer to use a function that has been specifically tailored to the machine we're using so that we won't have to worry about all these tedious details, and we won't have to modify our code every time we change machines. As you may have guessed, MPI provides such a function.

3.2 Broadcast

A communication pattern that involves all the processes in a communicator is a *collective communication.* As a consequence, a collective communication usually involves more than two processes. A *broadcast* is a collective communication in which a single process sends the same data to every process. In MPI the function for broadcasting data is MPI_Bcast:

```
int MPI_Bcast(void* message, int count,
    MPI_Datatype datatype, int root, MPI_Comm comm)
```

It simply sends a copy of the data in message on process root to each process in the communicator comm. It should be called by all the processes in the communicator with the same arguments for root and comm. Hence a broadcast message cannot be received with MPI_Recv. The parameters count and datatype have the same function that they have in MPI_Send and MPI_Recv: they specify the extent of the message. However, unlike the *point-to-point* functions, MPI insists that in collective communication count and datatype be the same on all the processes in the communicator [4]. The reason for this is that in some collective operations (see below), a single process will receive data from many other processes, and in order for a program to determine how much data has been received, it would need an entire *array* of return statuses.

We can rewrite the Get_data function using MPI_Bcast as follows.

```
void Get_data2(int my_rank, float* a_ptr, float* b_ptr,
    int* n_ptr) {
    int root = 0;  /* Arguments to MPI_Bcast */
    int count = 1;

    if (my_rank == 0)
```

```
    {
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    MPI_Bcast(a_ptr, 1, MPI_FLOAT, root,
        MPI_COMM_WORLD);
    MPI_Bcast(b_ptr, 1, MPI_FLOAT, root,
        MPI_COMM_WORLD);
    MPI_Bcast(n_ptr, 1, MPI_INT, root,
        MPI_COMM_WORLD);
} /* Get_data2 */
```

Certainly this version of Get_data is much more compact and readily comprehensible than the original, and if MPI_Bcast has been optimized for your system, it will also be a good deal faster.

3.3 Reduce

In the trapezoid rule program after the input phase, every processor executes essentially the same commands until the final summation phase. So unless our function $f(x)$ is fairly complicated (i.e., it requires considerably more work to evaluate over certain parts of $[a, b]$), this part of the program distributes the work equally among the processors. As we have already noted, this is *not* the case with the final summation phase, when, once again, process 0 gets a disproportionate amount of the work. However, you have probably already noticed that by reversing the arrows in figure 3.1, we can use the same idea we used in section 3.1. That is, we can distribute the work of calculating the sum among the processors as follows.

1. (a) 1 sends to 0, 3 sends to 2, 5 sends to 4, 7 sends to 6.
   (b) 0 adds its integral to that of 1, 2 adds its integral to that of 3, etc.
2. (a) 2 sends to 0, 6 sends to 4.
   (b) 0 adds, 4 adds.
3. (a) 4 sends to 0.
   (b) 0 adds.

Of course, we run into the same question that occurred when we were writing our own broadcast: is this tree structure making optimal use of the topology of our machine? Once again, we have to answer that this depends on the machine. So, as before, we should let MPI do the work, by using an optimized function.

The "global sum" that we wish to calculate is an example of a general class of collective communication operations called *reduction operations*. In a global reduction operation, all the processes (in a communicator) contribute data which is combined using a binary operation. Typical binary operations are addition, max, min, logical and, etc. The MPI function for performing a reduction operation is

```
int MPI_Reduce(void* operand, void* result,
    int count, MPI_Datatype datatype, MPI_Op op,
    int root, MPI_Comm comm)
```

MPI_Reduce combines the operands stored in *operand using operation op and stores the result in *result on process root. Both operand and result refer to count memory locations with type datatype. MPI_Reduce must be called by all processes in the communicator comm, and count, datatype, and op must be the same on each process.

The argument op can take on one of the following predefined values.

Operation Name	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical And
MPI_BAND	Bitwise And
MPI_LOR	Logical Or
MPI_BOR	Bitwise Or
MPI_LXOR	Logical Exclusive Or
MPI_BXOR	Bitwise Exclusive Or
MPI_MAXLOC	Maximum and Location of Maximum
MPI_MINLOC	Minimum and Location of Minimum

It is also possible to define additional operations. For details see [4].

As an example, let's rewrite the last few lines of the trapezoid rule program.

```
        ⋮
/* Add up the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);

/* Print the result */
     ⋮
```

Note that each processor calls MPI_Reduce with the same arguments. In particular, even though total only has significance on process 0, each process must supply an argument.

3.4 Other Collective Communication Functions

MPI supplies many other collective communication functions. We briefly enumerate some of these here. For full details, see [4].

- ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  MPI_Barrier provides a mechanism for synchronizing all the processes in the communicator comm. Each process blocks (i.e., pauses) until every process in comm has called MPI_Barrier.

- ```
  int MPI_Gather(void* send_buf, int send_count,
      MPI_Datatype send_type,  void* recv_buf,
      int recv_count, MPI_Datatype recv_type,
      int root, MPI_comm comm)
  ```

  Each process in comm sends the contents of send_buf to the process with rank root. The process root concatenates the received data in process rank order in recv_buf. That is, the data from process 0 is followed by the data from process 1, which is followed by the data from process 2, etc. The recv arguments are significant only on the process with rank root. The argument recv_count indicates the number of items received from each process — not the total number received.

- ```
  int MPI_Scatter(void* send_buf, int send_count,
      MPI_Datatype send_type, void* recv_buf,
      int recv_count, , MPI_Datatype recv_type,
      int root, MPI_Comm comm)
  ```

  The process with rank root distributes the contents of send_buf among the processes. The contents of send_buf are split into p segments each consisting of send_count items. The first segment goes to process 0, the second to process 1, etc. The send arguments are significant only on process root.

- ```
  int MPI_Allgather(void* send_buf, int send_count,
      MPI_Datatype send_type, void* recv_buf,
      int recv_count, MPI_Datatype recv_type,
      MPI_comm comm)
  ```

  MPI_Allgather gathers the contents of each send_buf on each process. Its *effect* is the same as if there were a sequence of p calls to MPI_Gather, each of which has a different process acting as root.

- ```
  int MPI_Allreduce(void* operand, void* result,
      int count, MPI_Datatype datatype, MPI_Op op,
      MPI_Comm comm)
  ```

  MPI_Allreduce stores the result of the reduce operation op in each process' result buffer.
  ```