Collective communication

There are many ways in which we can improve upon the trapezoid rule program.

One of the main ways we can improve upon the program is by confronting the way we are handling the global sum of computing all the trapezoids.

- One metaphor for this problem is:
  If we hire eight workers to, say, build a house, we might feel that we weren't getting our money's worth if seven of the workers told the first what to do, and then the seven collected their pay and went home.
  - But this is very similar to what we're doing in our global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing.
  - Sometimes this is the best case scenario but in this case we can do more to increase efficiency
- Tree-structured communication
  - We can have something like a binary tree structure where lets say processes 1,3,5, and 7 will send their values to 0,2,4,6 respectively and then those processes will do the sum. This process will repeat until we have our final area sum in process 0.

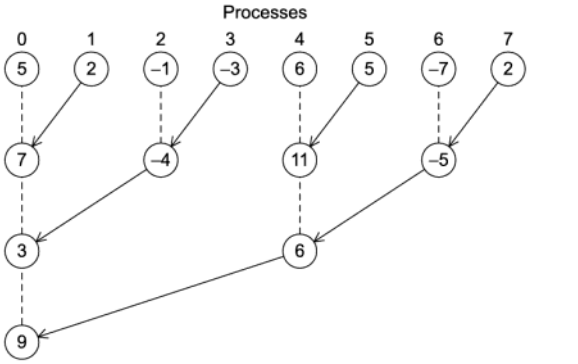


FIGURE 3.6
A tree-structured global sum.

  - This new scheme has the property that a lot of the work is done concurrently by different processes.
    - Not only do we compute the area as we go but we are also saving on calling MPI send and receive.
  - So we've reduced the overall time by more than 50%.

MPI_Reduce

- MPI provides a function who can do many of the "global sum" operations that we might need. The list of operations for a global sum are:

Table 3.2 Predefined Reduction Operators in MPI.

Operation Value	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

Note:
- If we want to find things like sum, maximum, minimum, product, true/false, etc

- We can use these in the function MPI_Reduce:

```
int MPI_Reduce(
      void*         input_data_p    /* in  */,
      void*         output_data_p   /* out */,
      int           count           /* in  */,
      MPI_Datatype  datatype        /* in  */,
      MPI_Op        operator        /* in  */,
      int           dest_process    /* in  */,
      MPI_Comm      comm            /* in  */);
```

- The 5th param (operator of type MPI_Op) is the one where we enter the operation that we want to use

- Some facts that we should know that distinguish functions like MPI_Send and MPI_Recv from a function like MPI_Reduce are:
  - Point to point vs collective communication
    - MPI_Send and MPI_Recv are what are known as point to point communications which say that they are processes who are talking directly to each other
    - MPI_Recv would be known as a collective communication which involves all the processes in the same communicator who are talking to each other
- Example:

The operator we want is `MPI_SUM`. Using this value for the `operator` argument, we can replace the code in Lines 18–28 of Program 3.2 with the single function call

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM,
      0, MPI_COMM_WORLD);
```

One point worth noting is that by using a `count` argument greater than 1, `MPI_Reduce` can operate on arrays instead of scalars. So the following code could be used to add a collection of N-dimensional vectors, one per process:

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Collective vs. point-to-point communications

3.4.3 Collective vs. point-to-point communications

It's important to remember that collective communications differ in several ways from point-to-point communications:

1. *All* the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
3. The `output_data_p` argument is only used on `dest_process`. However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.
4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags. So they're matched solely on the basis of the communicator and the *order* in which they're called. As an example, consider the calls to `MPI_Reduce` shown in Table 3.3. Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0. At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6. However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`. The *order* of the calls will determine the matching, so the value stored in `b` will be $1 + 2 + 1 = 4$, and the value stored in `d` will be $2 + 1 + 2 = 5$.

Table 3.3 Multiple Calls to `MPI_Reduce`.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

- Note that for point 4: we use table 3.3 to show that when we are making calls to MPI_Reduce it does not care about the addresses of the memory location instead it only cares about processes 0 output location for each MPI_Reduce call
  - What I mean by this is that for example the first MPI_Reduce call that process 0 makes is that the value in a be stored in b thus b = 0 + 1. At the same times process 1 calls for c's value to be added, we would assume that it would be added

to d but no it is actually added to b since process 0 is adding it to b, so b = 1 + 2. Now b is 3, and process 2 will add b = 3 + 1 making it 4, b = 4.

- The same thing would happen in the second call that process 0 does for MPI_Reduce. We will be adding to d instead of any other location it might seem like we are adding to.

Aliasing

A final caveat: it might be tempting to call `MPI_Reduce` using the same buffer for both input and output. For example, if we wanted to form the global sum of `x` on each process and store the result in `x` on process 0, we might try calling

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

However, this call is illegal in MPI. So its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash; it might even produce a correct result. It's illegal, because it involves **aliasing** of an output argument. Two

MPI_Allreduce

In our trapezoidal rule program, we just print the result. So it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which *all* of the processes need the result of a global sum to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Fig. 3.8). Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a *butterfly*. (See Fig. 3.9.) Once again, we don't want to have to decide on which structure to use, or how to code it for optimal performance. Fortunately, MPI provides a variant of `MPI_Reduce` that will store the result on all the processes in the communicator:

```
int MPI_Allreduce(
      void*         input_data_p   /* in  */,
      void*         output_data_p  /* out */,
      int           count          /* in  */,
      MPI_Datatype  datatype       /* in  */,
      MPI_Op        operator       /* in  */,
      MPI_Comm      comm           /* in  */);
```

The argument list is identical to that for `MPI_Reduce`, except that there is no `dest_process` since all the processes should get the result.

Broadcast

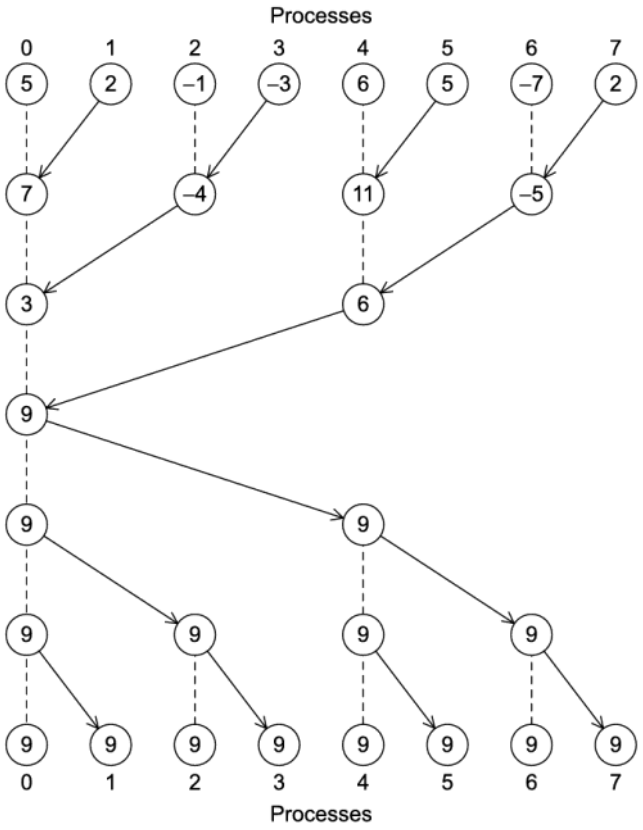
Similar to how we create a tree like structure for ouputs where we compute for some output, we can do the same for taking in some input and distributing it to all the processes.

- This is in fact what part of our helper function get_input(..) does
- How can we make it more efficient to distribute the input to all the processes?
- MPI provides a functions MPI_Bcast to do just this:

```
int MPI_Bcast(
      void*         data_p         /* in/out */,
      int           count          /* in     */,
      MPI_Datatype  datatype       /* in     */,
      int           source_proc    /* in     */,
      MPI_Comm      comm           /* in     */);
```

The process with rank source_proc sends the contents of the memory referenced by data_p to all the processes in the communicator comm.

- Some program, like our trapizoidal rule prg will look like this:



Global sum followed by a distribution of all of the sums

- We can modify our get_input(..) function to broadcast all the info we need to broadcast out to all the processes

```
1  void Get_input(
2        int       my_rank  /* in  */,
3        int       comm_sz  /* in  */,
4        double*   a_p      /* out */,
5        double*   b_p      /* out */,
6        int*      n_p      /* out */) {
7
8     if (my_rank == 0) {
9        printf("Enter a, b, and n\n");
10       scanf("%lf %lf %d", a_p, b_p, n_p);
11    }
12    MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13    MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
14    MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
15 }  /* Get_input */
```

Program 3.6: A version of `Get_input` that uses `MPI_Bcast`.