Sending messages around a ring

Dr. Christopher Marcotte — *Durham University*

In this exercise we'll write a simple form of global reduction. We will set up the processes in a ring (so each process has a left and right neighbour) and each process should initialise a buffer to the corresponding rank. To compute a global summation a simple method is to rotate each piece of data all the way round the ring: at each step, a process receives from the left and sends to the right. This is illustrated for 4 processes in the figure below. The accumulation happens on each rank (the boxes) – signifying the local sum on each rank – and the message at each step is shown by the arrow between processes, with the content of the message highlighted.

Each rank sends along the most recent value it's recieved, at each iteration, beginning with it's own value.

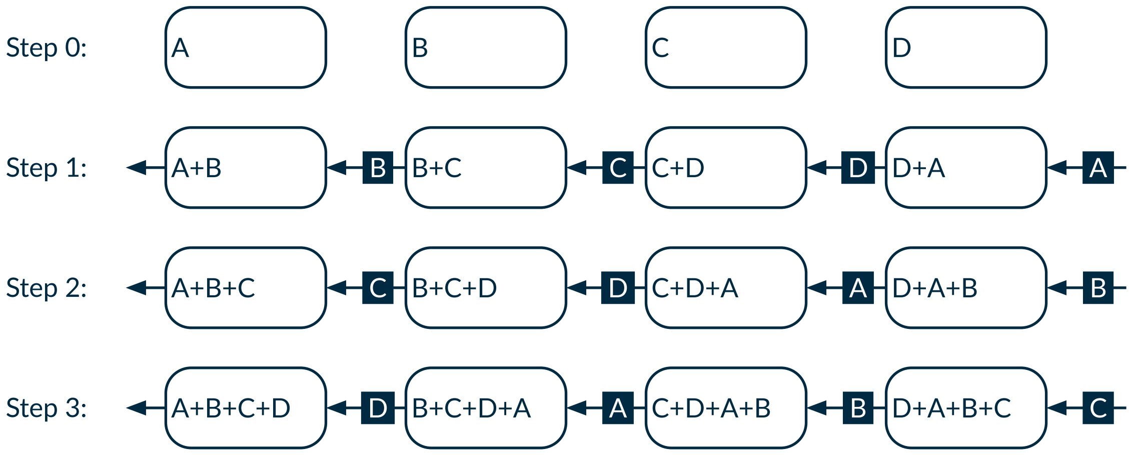

Figure 1: Rotating a message around a ring of processes, and accumulating, to produce an `Allreduce`.

`Allreduce`

Consider the template code below; it initialises and finalises MPI for a ring-allreduce, which you should implement.

ring.c

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv){
  MPI_Init(&argc, &argv);

  MPI_Comm comm;
  comm = MPI_COMM_WORLD;

  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  int local_value = rank;
  int summed_value = 0; // initialized sum: 0

  printf("[%d] Before reduction: local value is %d; summed value is %d\n",
        rank, local_value, summed_value);
  double start = MPI_Wtime();
  /*
  // Implement the ring-reduction here
  */
  double end = MPI_Wtime();
  printf("[%d] After reduction: local value is %d; summed value is %d (should be %d)\n",
        rank, local_value, summed_value, size*(size-1)/2);
  if (rank == 0){
    printf("\tReduction on %d processes took %e seconds.\n", size, end-start);
  }
  MPI_Finalize();
  return 0;
}
```

The initial local value is set to the rank. This gives us a nice way of checking if the summed value is correct, since on P processes, the final value should be $\frac{P(P-1)}{2}$.

Exercise

Implement a ring reduction for buffers of a fixed size (as in the code stub provided). Ensure that your code produces the correct answer for any number of processes. That is, do not hard-code the total number of processes anywhere.

Hint

If you're already comfortable with non-blocking messages, you can use those, otherwise, you will probably find `MPI_Sendrecv` useful.

Solution

The relevant part of a solution using `MPI_Sendrecv` is shown below. This uses a third buffer to receive the local value into, before performing the swap with the current rank local value. If you use `&local_value` in both the `*recvbuf` and `*sendbuf` positions of the MPI call, you may get non-deterministic results.

ring-solution.c

```c
  for (int n = 0; n < size; n++){
    summed_value += local_value;
    MPI_Sendrecv( &local_value, 1, MPI_INT, (rank + 1) % size, rank,
                  &recv_value, 1, MPI_INT, (rank - 1 + size) % size, (rank - 1 + size) % size,
                  comm, MPI_STATUS_IGNORE);
    local_value = recv_value;
    /*
    // You may find this print statement helpful for understanding the progression
    if (rank == 0){
      printf("\t[%d] Step %d: local value is %d; summed value is %d\n", rank, n, local_value, summed_value);
    }
    */
  }
```

An important feature of using `MPI_Sendrecv` here is that we need not worry about deadlocks. If you spent some time googling for a solution, you probably stumbled on this example of a ring-reduction, which must manually manage some subtle ordering for the `MPI_Send` and `MPI_Recv` calls to avoid a deadlock, due to the blocking nature of `MPI_Send` and `MPI_Recv`.

If you use a non-blocking send and recieve, e.g. `MPI_Isend` and `MPI_Irecv`, you can avoid deadlocks, as well; these methods are more useful for less regular patterns than the ring sharing we are using in this example, however.

Scaling study: adding processes

Having implemented the reduction, we'll now measure how the performance varies when we increase the number of processes. First, think about how many messages your program sends in total as a function of the total number of processes.

If each message takes a constant amount of time, what algorithmic complexity do you expect for your implementation as a function of the total number of processes P?

Exercise

Use the batch system to time the performance of your reduction for increasing numbers of processes on Hamilton.

Record the runtime for the reduction for each P, and produce a plot of the runtime as a function of P. Does it align with your expectations?

If the total time if *very* small, you might need to run the reduction in a loop to get reasonable measurements – see earlier timing approaches from the array access exercise with OpenMP for inspiration.

Hint

You can time the reduction code with `MPI_Wtime()`, like so:

ring.c

```c
double start = MPI_Wtime();
/*
// Implement the ring-reduction here
*/
double end = MPI_Wtime();
```

Solution

The ring reduction should take $T_{\text{ring}}(P) \propto P$ time to complete with increasing P. Alternative all-reductions, e.g., `MPI_Allreduce`, use a tree algorithm by default. Trees will take $T_{\text{tree}}(P) \propto \log_2 P$, so for large P will be *much* faster. For small P, it is less clear. In the plot below, the two curves intersect, and the first $P > 1$ where $T_{\text{tree}}(P') < T_{\text{ring}}(P')$ for all $P' > P$ is $P \approx 7$.

Likewise, we show the timing results of the ring-allreduce – due to some clever compilation effects, the efficiency for small numbers of processes

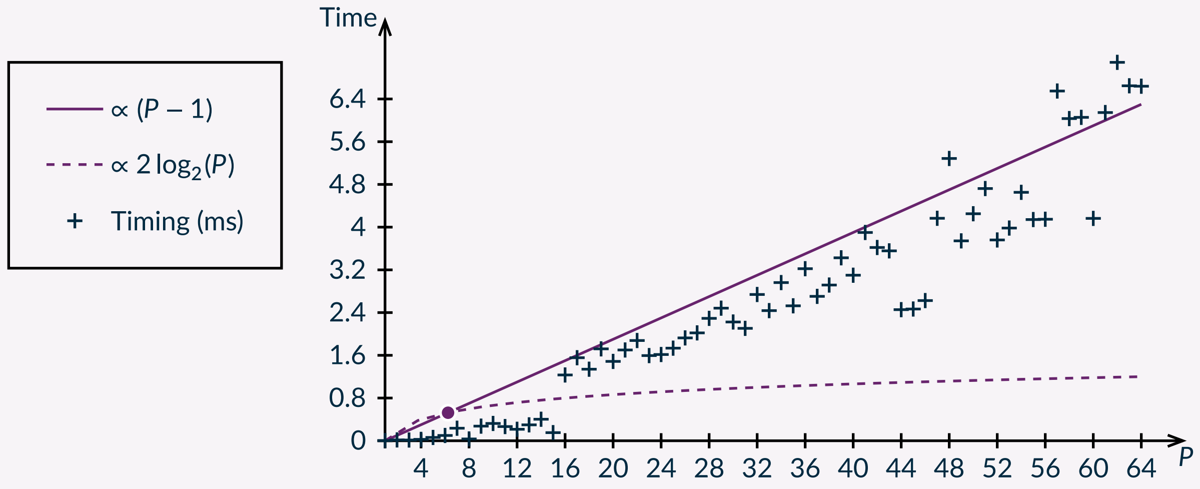

Figure 2: Scaling of ring allreduce as a function of P.

Comparison with Shared Memory parallelism In OpenMP, we typically employ `#pragma omp reduction(op:val)` to perform a reduction on `op` into `val`. This works by sharing state across threads by reading and writing from memory. If used carelessly, this sometimes results in data races which yield unreliable answers.

The approach in MPI is to have no shared memory access[*], and instead expect all sharing to occur through message passing. This approach is far more explicit, which avoids some issues (data races) while putting the onus on the programmer (explicit message passing).

Challenge

Consider a computational problem and think critically about what aspects make shared memory parallelism with OpenMP or distributed memory parallelism with MPI a better fit. Explain why you might prefer shared memory parallelism for problems with a lot of reductions, or why you might prefer distributed memory parallelism for largely compute-bound problems.

Aims

- Introduction to collective operations with a manual implementation
- A consideration of different communication patterns and their impact on performance for scaling
- An exploration of archetypal reductions.

[*]Save for the memory sharing constructs in MPI-3.