

MPI deadlocks and blocking

Dr. Christopher Marcotte — Durham University

MPI and deadlocks

In this exercise we will modify a simple code that sends a message between two processes.

In this directory you will find `ptp-deadlock.c`, and the relevant bit of code is this piece, which sends and receives a message between processes:

`ptp-deadlock.c`

```
26  if (rank == 0) {
27      MPI_Send(sendbuf, nentries, MPI_INT, 1, 0, MPI_COMM_WORLD);
28      MPI_Recv(recvbuf, nentries, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
29  } else if (rank == 1) {
30      MPI_Send(sendbuf, nentries, MPI_INT, 0, 0, MPI_COMM_WORLD);
31      MPI_Recv(recvbuf, nentries, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32  }
```

Recall that `MPI_Send(...)` behaves like `MPI_Bsend(...)` when there is available buffer space, and behaves like `MPI_Ssend(...)` when there is not, blocking for the corresponding `MPI_Recv(...)`.

Exercise

Compile the program `ptp-deadlock.c` using

```
mpicc -o ptp-deadlock ptp-deadlock.c
```

and run the program, supplying `nentries` as an integer command line argument:

```
mpirun -n 2 ./ptp-deadlock <nentries>
```

Working from 1 in powers of 2, how large can you make `<nentries>` before the program deadlocks?

Hint

You will need `Control-c` (or `scancel <jobid>` on the batch system) to end the job once it deadlocks.

Exercise

Modify the code to use `MPI_Ssend` instead of `MPI_Send`. How large can you make `nentries` now?

Exercise

Modify the code to use `MPI_Sendrecv(...)` instead of `MPI_Ssend(...)` and `MPI_Recv(...)`.

Challenge

Instead, use `MPI_Bsend` and set up the necessary buffers. Does this deadlock?

Global Gather

In this exercise we'll implement a rudimentary collective operation, where we gather some information from each process onto rank 0, and fill a buffer. This involves a lot of communication, and so gives an opportunity to test latency and throughput of some blocking and non-blocking MPI calls.

Exercise

Complete an MPI code in which rank-0 gathers a message from every process and places it in an array at a position corresponding to the rank of the sender. Implement two functions, `gather_blocking(...)` and `gather_nonblocking(...)`, so you can test their performance.

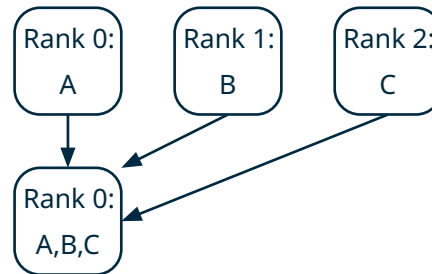


Figure 1:

See the `int main(...)` function in `gather-from-all.c`:

gather-from-all.c

```

39 int main(int argc, char **argv){
40     MPI_Init(&argc, &argv);
41
42     MPI_Comm comm = MPI_COMM_WORLD;
43     int rank, size;
44     MPI_Comm_rank(comm, &rank);
45     MPI_Comm_size(comm, &size);
46
47     int local;
48     local = rank * rank;
49
50     int *blocking = NULL;
51     int *nonblocking = NULL;
52     if (rank == 0) {
53         /* Allocate space for output arrays -- 1 int per process. */
54         blocking = malloc(size*sizeof(*blocking));
55         nonblocking = malloc(size*sizeof(*nonblocking));
56     }
57     double start, end;
58     start = MPI_Wtime();
59     for (int i = 0; i < 100; i++) {
60         gather_blocking(&local, blocking, comm);
61     }
62     end = MPI_Wtime();
63     if (rank == 0) {
64         printf("Blocking gather takes %.3g s\n", (end - start)/100);
65     }
66     start = MPI_Wtime();
67     for (int i = 0; i < 100; i++) {
68         gather_nonblocking(&local, nonblocking, comm);
69     }
70     end = MPI_Wtime();

```

```
71  if (rank == 0) {
72      printf("Non-blocking gather takes %.3g s\n", (end - start)/100);
73  }
74  free(blocking);
75  free(nonblocking);
76  MPI_Finalize();
77  return 0;
78 }
```

We allocate space for P processes outputs (a single `int` each) on rank 0, and then time how long it takes to call the two gathering functions a fixed number of times (in this case, 100, arbitrarily). Compare the timing of the two versions. Which performs better? Does it depend on the total number of messages, P ?

Hint

Process 0 uses a blocking `MPI_Recv(...)` for all receives in `gather_blocking(...)`, and uses a non-blocking `MPI_Irecv(...)` followed by `MPI_Waitall(...)` in `gather_nonblocking(...)`.

Challenge

Modify the code to compute the average time for these exchanges, over multiple instances — this will give more reliable results for such short times.

Aims

- Introduction to MPI deadlocks and how they impact program structure
- Understanding basics of blocking communications
- Understanding the primary failure mode of MPI