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 recieves a message between processes:

ptp-deadlock.c

```c
26   if (rank == 0) {                                                                    c
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, suppling `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.

Solution

This depends on your machine. On my laptop it is 1024 entries.

Exercise

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

Solution

The calling syntax is the same, so just add that extra `s` in the function name:

ptp-deadlock-solution.c

```c
34   if (rank == 0) {
35     MPI_Ssend(sendbuf, nentries, MPI_INT, 1, 0, MPI_COMM_WORLD);
36     MPI_Recv(recvbuf, nentries, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
37   } else if (rank == 1) {
38     MPI_Ssend(sendbuf, nentries, MPI_INT, 0, 0, MPI_COMM_WORLD);
39     MPI_Recv(recvbuf, nentries, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
40   }
```

And, of course, it deadlocks for any size message because both processes are waiting inside `MPI_Ssend(...)` for `MPI_Recv(...)`s that will never come.

Exercise

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

Solution

ptp-deadlock-solution.c

```c
42   if (rank == 0) {
43     MPI_Sendrecv( sendbuf, nentries, MPI_INT, 1, 0, /* Send parameters */
44                   recvbuf, nentries, MPI_INT, 1, 0, /* Recv parameters */
45                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
46   } else if (rank == 1) {
47     MPI_Sendrecv( sendbuf, nentries, MPI_INT, 0, 0, /* Send parameters */
48                   recvbuf, nentries, MPI_INT, 0, 0, /* Recv parameters */
49                   MPI_COMM_WORLD, MPI_STATUS_IGNORE);
50   }
```

This code completes for any size message (up to some very unrealistically large sizes); `nentries=1073741824` (2^{30}) took a few seconds but completed.

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.

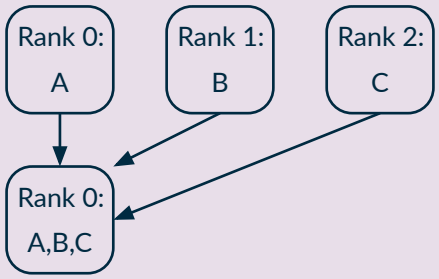

Figure 1:

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

gather-from-all.c

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

  MPI_Comm comm = MPI_COMM_WORLD;
  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);

  int local;
  local = rank * rank;

  int *blocking = NULL;
  int *nonblocking = NULL;
  if (rank == 0) {
    /* Allocate space for output arrays -- 1 int per process. */
    blocking = malloc(size*sizeof(*blocking));
    nonblocking = malloc(size*sizeof(*nonblocking));
  }
  double start, end;
  start = MPI_Wtime();
  for (int i = 0; i < 100; i++) {
    gather_blocking(&local, blocking, comm);
  }
  end = MPI_Wtime();
  if (rank == 0) {
    printf("Blocking gather takes %.3g s\n", (end - start)/100);
  }
  start = MPI_Wtime();
  for (int i = 0; i < 100; i++) {
    gather_nonblocking(&local, nonblocking, comm);
  }
  end = MPI_Wtime();
  if (rank == 0) {
    printf("Non-blocking gather takes %.3g s\n", (end - start)/100);
  }
  free(blocking);
  free(nonblocking);
  MPI_Finalize();
  return 0;
}
```

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(...)`.

Solution

In both functions, we switch to different behavior for rank 0, first inserting the rank 0 value into index 0, and either creating the requests, immediate-recieving, and waiting – or using a blocking recieve. Note that in both cases, we use `&(recvbuf[i])` for the recieve buffer so the result is offset into the allocated array.

gather-from-all-solution.c

```c
void gather_nonblocking(const int *send, int *recvbuf, MPI_Comm comm){
  int size, rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);

  if (rank == 0) {
    MPI_Request *requests;
    requests = malloc((size-1) * sizeof(*requests));
    recvbuf[0] = send[0];
    for (int i = 1; i < size; i++) {
      MPI_Irecv(&(recvbuf[i]), 1, MPI_INT, i, 0, comm, &(requests[i-1]));
    }
    MPI_Waitall(size-1, requests, MPI_STATUSES_IGNORE);
    free(requests);
  } else {
    MPI_Send(send, 1, MPI_INT, 0, 0, comm);
  }
}

void gather_blocking(const int *send, int *recvbuf, MPI_Comm comm){
  int size, rank;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);

  if (rank == 0) {
    /* Put my value in the first spot */
    recvbuf[0] = *send;
    for (int i = 1; i < size; i++) {
      /* Receive from each process in turn */
      MPI_Recv(&(recvbuf[i]), 1, MPI_INT, i, 0, comm, MPI_STATUS_IGNORE);
    }
  } else {
    /* Send to rank 0 */
    MPI_Send(send, 1, MPI_INT, 0, 0, comm);
  }
}
```

On my laptop, I get roughly a factor of 10 × between the versions:

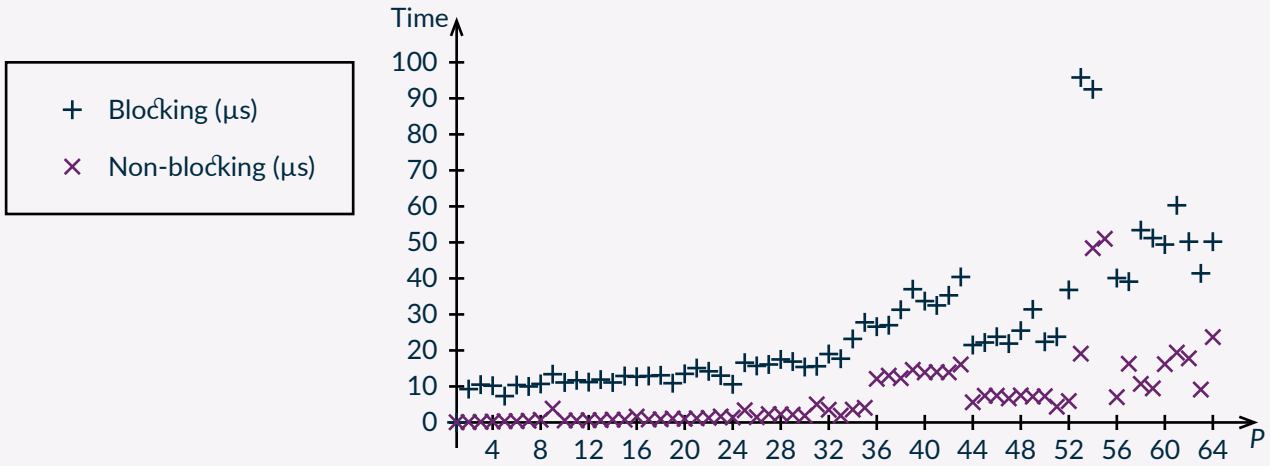

Figure 2: Scaling of ring allreduce as a function of *P*.

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