High-Performance Computing: MPI (ctd)

Adrian F. Clark: alien@essex.ac.uk

2015–16

A reminder

Last time, we started looking at MPI, the **M**essage **P**assing **I**nterface

This time, still in the context of MPI, we shall first look at one of the major problems with multi-process computing; then we shall take a look at different ways of organising processes on processors

Exchanging data between processes

What is the problem with the following?

```
if (rank == 0) {
  MPI_send (buf1, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, err);
  MPI_recv (buf2, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
    status, err);
} else {
    MPI_send (buf2, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, err);
    MPI_recv (buf1, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
      status, err);
}
```

The problem is that the processes can — and that means probably will over time — end up in *deadlock*, in which each process has sent a message and is waiting for the other to receive it

There are several requirements for deadlock to occur:

1. resources may not be shared
2. processes may hold resources while requesting other resources
3. resources may not be pre-empted while in use
4. a circular chain of processes exists, each holding a resource requested by the next process in the chain

Removing any of these requirements will avoid (or break) deadlock

There are three ways around the problem

The first is to use a *buffered* send, though it involves making a copy of the data to be transferred and so is inefficient for large datasets

```
if (rank == 0) {
  MPI_Bsend (buf1, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, err);
  MPI_recv (buf2, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
    status, err);
} else {
    MPI_Bsend (buf2, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, err);
    MPI_recv (buf1, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
      status, err);
}
```

The second solution is to use a *non-blocking* send, though the program then has to wait until the data have been copied from the buffer

```
if (rank == 0) {
  MPI_Isend (buf1, 1, MPI_INT, 1, 10, MPI_COMM_WORLD,
    REQ, err);
  MPI_recv (buf2, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
    status, err);
} else {
    MPI_Isend (buf2, 1, MPI_INT, 0, 20, MPI_COMM_WORLD,
        REQ, err);
    MPI_recv (buf1, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
      status, err);
}
MPI_Wait (REQ, status);  // wait until send completes
```

However, the best solution is to exchange the order of MPI_Send and MPI_Recv calls in *one* process

You must not change the order in *both* processes or you will end up with potential deadlock again

```
if (rank == 0) {
  MPI_send (buf1, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, err);
  MPI_recv (buf2, 1, MPI_INT, 1, 20, MPI_COMM_WORLD,
    status, err);
} else {
    MPI_recv (buf1, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
      status, err);
    MPI_send (buf2, 1, MPI_INT, 0, 20, MPI_COMM_WORLD, err);
}
```

Collective communication

`MPI_Send` and `MPI_Recv` perform point-to-point communication, *i.e.*, they involve a single sender and a single receiver

MPI provides routines for performing *collective* communication: one-to-many, many-to-one, many-to-many — we shall look at some (though not all) of these

Synchronisation

As MPI_Send and MPI_Recv are blocking, they inherently cause the calling and receiving processes to synchronise

When it is necessary for *all* processes in a group to be synchronised, they should all invoke MPI_Barrier when they have reached the point at which the synchronisation is needed

Only when all the processes have invoked MPI_Barrier will any of them be able to continue execution

Broadcast

A common programming idiom is for the master process to propagate information to all the workers

You could program this explicitly using a set of `MPI_Send` invocations, as in **mpi-04.c**, but there is a shorter (and usually faster) way:

```
int MPI_Bcast (void *message, int count,
   MPI_Dataype type, int root, MPI_Comm comm)
```

This should be called by *all* processes in `comm` with the same arguments for `root` and `comm` (and `count` and `type`)

It sends a copy of the data in `message` on process `root` to all the processes in `comm` — see **mpi-05.c**

Scatter and Gather

MPI_Bcast is fine if all processes are to receive exactly the same data; but it is normal for each worker to process separate data distributed by the farmer

The easiest way to do this is to load *all* the data into the farmer as a single array and then distribute it using MPI_Scatter (or MPI_Scatterv for variable-sized chunks):

```
int MPI_Scatter(void *sendbuf, int sendcnt,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcnt, MPI_Datatype recvtype, int root,
    MPI_Comm comm)
```

where sendcnt is the number of elements of sendbuf to send to each process

The converse of MPI_Scatter is MPI_Gather, in which a process collects data from all other processes in the communicator

```
int MPI_Gather(void *sendbuf, int sendcnt,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcnt, MPI_Datatype recvtype,
    int root, MPI_Comm comm)
```

As with MPI_Bcast, all processors in the communicator make the same call to MPI_Scatter and MPI_Gather (that's why they both require the "root" process as a parameter)

In MPI_Scatter, sendbuf, sndcnt and sendtype make sense only in the root process; likewise recvbuf, recvcnt and recvtype in MPI_Gather

See **mpi-07.c**

Careful use of these routines can reduce data movement

N processes, if all workers are sent messages by the farmer, there will be $N - 1$ messages

However, if one builds a 'communication tree' and uses `MPI_Bcast`, this can be reduced to $O(\log N)$ messages

Collective Computation

`MPI_Reduce` applies some computation to an operand in every participating process

For example, adding together a `float` from every process and putting the result in a process

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

The routine is called by all group members using the same arguments for `count`, `datatype`, `op`, `root`, and `comm`

Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type

This is a particularly elegant solution when many processes perform a partial computation that need to be combined to give a final result

Examples of MPI_Op operations:

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and

Topologies

There are many problems for which a solution is most easily constructed when the processes can be thought of as having a particular topology

In scientific computing, the most common topology is a *mesh* (in N dimensions); for other topologies, `MPI_Create_graph` can be used to construct them

Why are topology routines in MPI? They allow MPI implementations to exploit hardware features that allow logical neighbours to be electronically adjacent, thereby speeding up transfers

Sketch out an MPI program that passes a message around a ring of processes of size N

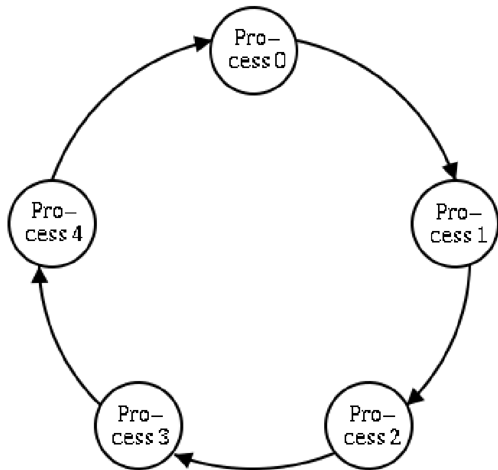

Figure 1:Ring of processes

Mesh ('Cartesian') topologies

We can set up a 2D mesh of 3×4 processes as follows:

```
#define NDIMS 2
int dims[NDIMS] = {3, 4};
int periods[NDIMS] = {False, False};
int reorder = True;
MPI_Comm *comm2d;
MPI_Cart_create (MPI_COMM_WORLD, NDIMS, dims, periods,
  reorder, comm2d, err);
```

This creates a new communicator with the same processes but having a mesh virtual topology

The `periods` array specify what happens at the edges of the mesh:

- If an element is set to `False`, the processes at the edge of the mesh have no neighbour (and asking for it will receive `MPI_NULL_RANK`)
- If an element is set to `True`, the neighbour beyond the highest-numbered process in a particular rank is the lowest-numbered process (*cyclic wrap-around*)

A process can find where it lies in the array using

```
MPI_Comm comm;
int rank;
int coords[NDIMS];
int MPI_Cart_coords (comm, rank, NDIMS, coords);
```

See **mpi-06.c**

MPI_Cart_shift can be used to obtain the source and destination ranks of the calling process, me, resulting from shifting along the first direction of the 2D cartesian grid by one

```
// Create cartesian topology
period[0] = 1;      // cyclic in this direction
period[1] = 0;      // not cyclic in this direction
MPI_Cart_create (MPI_COMM_WORLD, NDIMS, dims, period, reorder,
MPI_Comm_rank (comm2D, &me);
MPI_Cart_coords (comm2D, me, NDIMS, coords);

source = me;   // calling process rank in 2D communicator
index =  0;    // shift along the 1st index (out of 2)
displ =  1;    // shift by 1
MPI_Cart_shift (comm2D, index, displ, source, &dest1);
```