6. Collective Communication

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

Goals

- Continue our survey of MPI by considering collective communications – an important part of the API.
- In several cases, motivate introduction of these methods by illustrating how their use can simplify parts of programs introduced in earlier labs.

Collective Communication

- Past two lectures introduced fundamental methods for communication between processes of a distributed memory parallel program.
- These generally took the form of *point-to-point communication* – messages sent between *two* processes.
- A different and important paradigm (supported by MPI) involves *all* processes working together to move data between the memories of the processors.
- This is called *collective communication*.

BROADCAST – THE SIMPLEST COLLECTIVE

Idea of Broadcast

- The idea is fairly intuitive.
- One process needs to send a particular data item (embodied in MPI in an array) to *every other process* in the program.
- Very commonly the broadcasting process may be *process 0*, and the broadcast data may be some input data, or values that control the program as a whole
  - e.g. the size, N, of the problem to be solved, where this isn't declared globally as a constant.

Broadcast using Send/Recv

```
int [] values = new int [M] ;
if (me == 0) {
    ... Initialize `values' - e.g. input them from user ...
    for (int dst = 1 ; dst < P ; dst++) {
        MPI.COMM_WORLD.Send(values, 0, M, MPI.INT, dst, 0) ;
    }
}
else {      // me > 0
        MPI.COMM_WORLD.Recv(values, 0, M, MPI.INT, 0, 0) ;
}
... Consistent values now available to all processes ...
```

Critique

- This works fine, but requires the programmer to think about the details of how the broadcast is broken down into sends and receives.
- Moreover this implementation is *less efficient* than it needs to be – it probably takes time $O(M \times P)$ to complete.
  - There are much more efficient algorithms to implement a broadcast, but their logic is more complex.

Broadcast using a Collective

```
int [] values = new int [M] ;
if (me == 0) {
    ... Initialize `values' - e.g. input them from user ...
}

MPI.COMM_WORLD.Bcast(values, 0, M, MPI.INT, 0) ;

... Consistent values now available to all processes ...
```

The Bcast Method

- The new method looks like this:

  Bcast(buffer, offset, count, type, root)

  where buffer, offset, count and type describe source and destination arrays (as previously) and root is rank of broadcasting process.

- Importantly, **Bcast** must be called by *all processes*, "at the same point" in a program, and with *consistent arguments* (e.g. all must agree on the values of root, type, etc).
  - Recalls usage of *barrier* in shared memory programs.

Advantages of Bcast

- The user code is shorter, because the logic of the communication pattern is captured in the library.
- Perhaps more importantly it should be faster – a well-tuned implementation of Bcast may complete in time $O(M + \log(P))$ for large messages and numbers of processors.
- Broadcast is an archetype for a whole family of collective communications.

REDUCTION

Reduce vs Broadcast

- *Reduction* is in a sense the opposite operation to broadcast.
- Here values are taken from *all* processes, reduced to *single values* by some *combining operation*, and those single values are deposited on a "root" process.
- We have already seen this kind of pattern in our first MPJ code for calculating π, where the combining operation was floating point addition.

MPJ Parallel π Collecting Results

```
if (me > 0) {
    double [] sendBuf = new double [] {sum} ;
            // 1-element array containing sum

    MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 0) ;
}
else {        // me == 0 !

    double [] recvBuf = new double [1] ;

    for (int src = 1 ; src < P ; src++) {

        MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 0) ;

        sum += recvBuf [0] ;
    }
}

double pi = step * sum ;
```

Collecting π Results using Reduce

```
double [] sendBuf = new double [] {sum} ;
              // 1-element array containing local sum

double [] recvBuf = new double [1] ;

MPI.COMM_WORLD.Reduce(sendBuf, 0, recvBuf, 0,
                                            1, MPI.DOUBLE, MPI.SUM, 0) ;

if (me == 0) {
    double pi = step * recvBuf [0] ;
    ...
}
```

Reduce Interface

- MPJ interface looks like:

  Reduce(sendbuf, sendoffset, recvbuf, recvoffset,
  count, type, op, root)

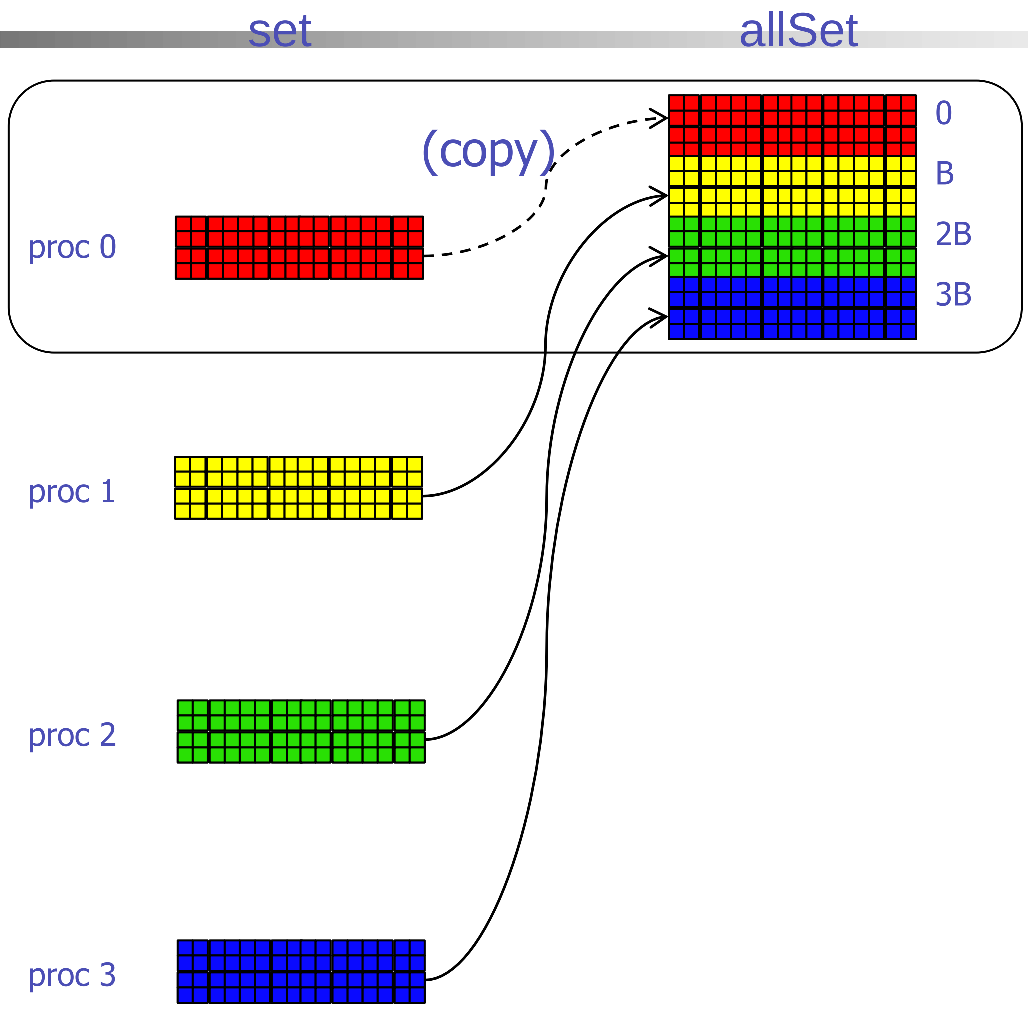
- If count is > 1, sendbuf arrays from *P* processes are combined element by element to produce an array of count results in recvbuf.
- op is the combining operation, and it can take following values:
  - MPI.SUM, MPI.PROD, MPI.MAX, MPI.MIN, MPI.LAND, MPI.BAND, MPI.LOR, MPI.BOR, MPI.LXOR, MPI.BXOR, MPI.MINLOC and MPI.MAXLOC

GATHER AND SCATTER

MPJ Mandelbrot Revisited

- In the week 5 lecture, we sketched one possible implementation of a Mandelbrot Set calculation using MPJ.
- The only non-trivial communication required there was collecting together results at the end of the calculation, reproduced below.

Communication Pattern



Gathering Results in Mandelbrot

```
if(me > 0) {
    MPI.COMM_WORLD.Send(set, 0, B, MPI.OBJECT, 0, 0) ;
}
else {    // me == 0

    for(int i = 0 ; i < B ; i++) {    // copy local `set' to start of `allSet'
        for(int j = 0 ; j < N ; j++) {
            allSet [i] [j] = set [i] [j] ;
        }
    }

    for(int src = 1 ; src < P ; src++) {
        MPI.COMM_WORLD.Recv(allSet, src * B, B, MPI.OBJECT, src, 0) ;
    }
     ... display allSet ...
}
```

The "gather" operation

- This behaviour is captured in the general "gather" operation – one of the collective operations supported directly in MPI.
- In MPJ the interface is fairly complex because it involves two separate arrays:

```
Gather(sendbuf, sendoffset, sendcount, sendtype,
       recvbuf, recvoffset, recvcount, recvtype,
       root)
```

Results in Mandelbrot using Gather

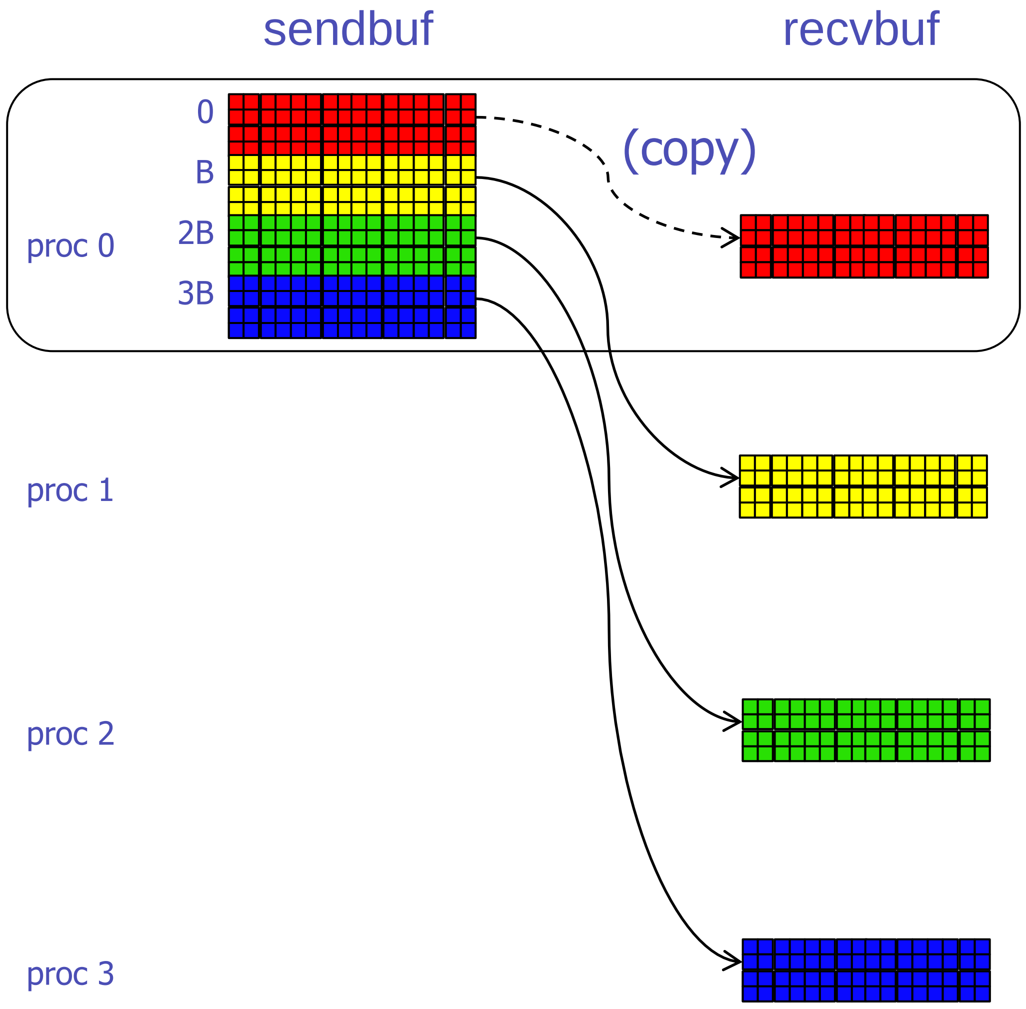
```
MPI.COMM_WORLD.Gather(set, 0, B, MPI.OBJECT,
                      allSet, 0, B, MPI.OBJECT,
                      0) ;
if(me == 0) {
    ... display allSet ...
}
```

- Results sent from P processes automatically get concatenated together into $P \times B$ elements of the recvbuf array, where B is the value of recvcount argument.
- In obscure cases sendcount and recvcount could be different, or sendtype and recvtype could be different. Usually, as here, they take the same values.

Scatter

- *Scatter* is the opposite operation to gather.
- A common scenario is where process 0 (say) initializes the values in a large array (e.g. by reading from a file), then these values have to be distributed across all processes for data-parallel processing.
- In MPJ, argument list of Scatter is identical to Gather, but now the first argument sendbuf is the "large" array that will be divided up into *B*-sized chunks.

Scatter Communication Pattern



(Assume root is 0.)

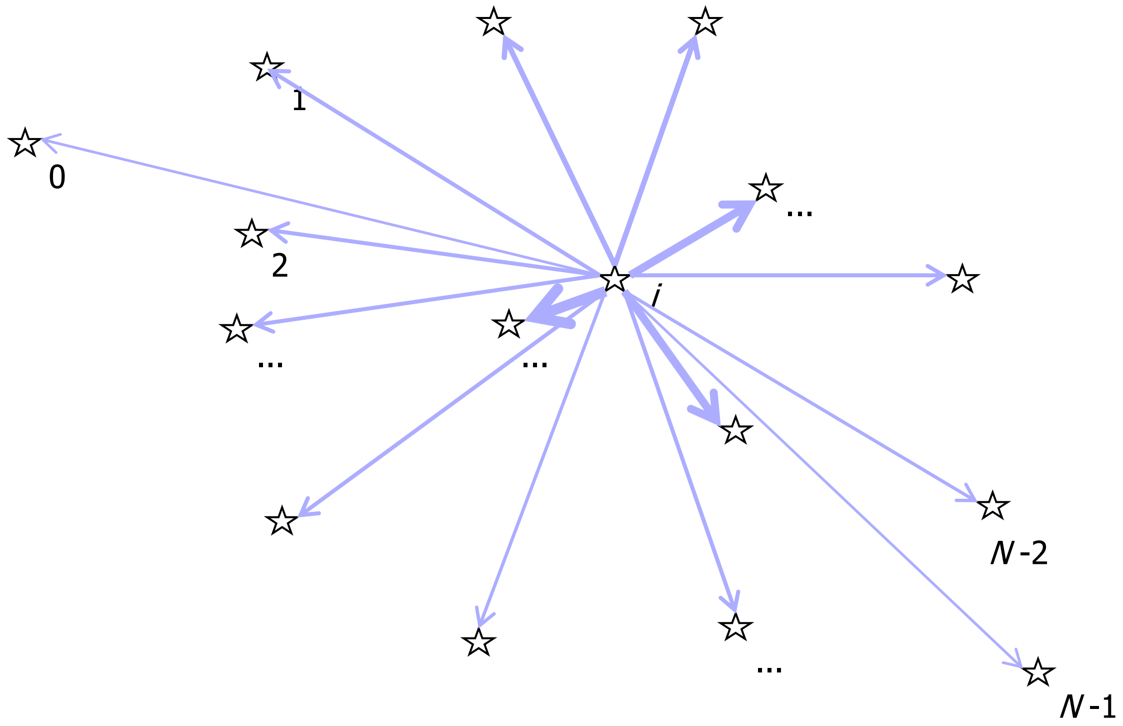
(Again assume 2d arrays – not necessary!)

Allgather

- The MPI Allgather operation behaves like a gather followed by a broadcast.
- As a motivating example, consider the simulation of *N* stars in a galaxy moving under the force of gravity.
- Each individual star feels the gravitational force of every other star, according to *Newton's inverse square* law of gravity.

Forces on stars

- Each star (one selected here) feels force of gravity from every other star.



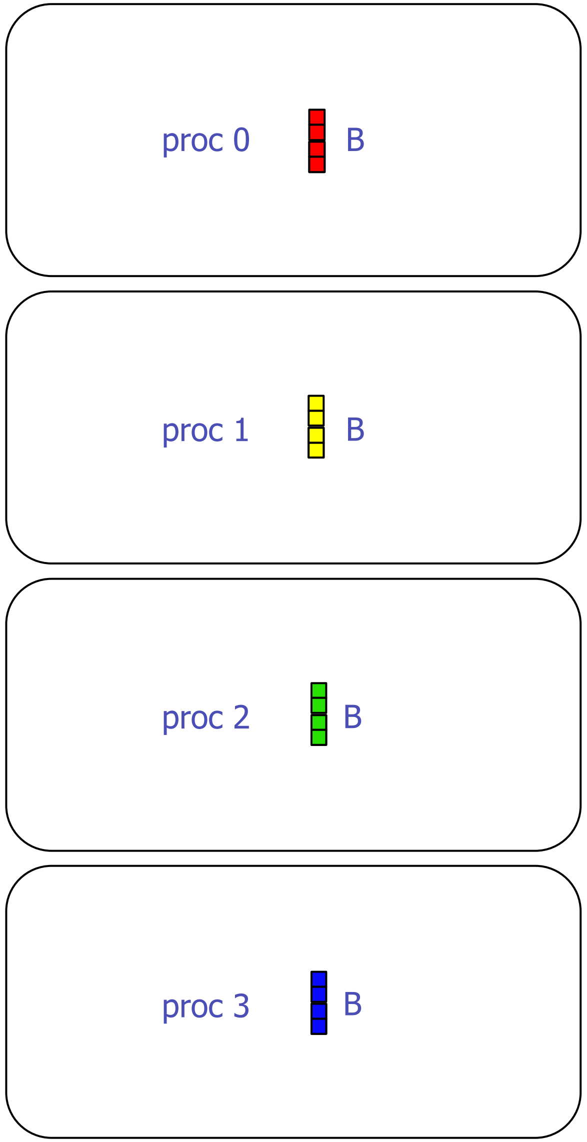
Total Force on a star

- Force on the i'th star:

$$\mathbf{F}_i = \sum_{j \neq i} \frac{G\mathbf{n}_{ij}}{(\mathbf{r}_i - \mathbf{r}_j)^2}$$

- Here $\mathbf{r}_i$ is (3-vector) position of i'th star and $\mathbf{n}_{ij}$ is a vector that takes into account the direction of the individual force (along the line from *i* to *j*).
- Don't worry about mathematical details! I have assumed all stars have same mass.

Decomposition of Star Positions

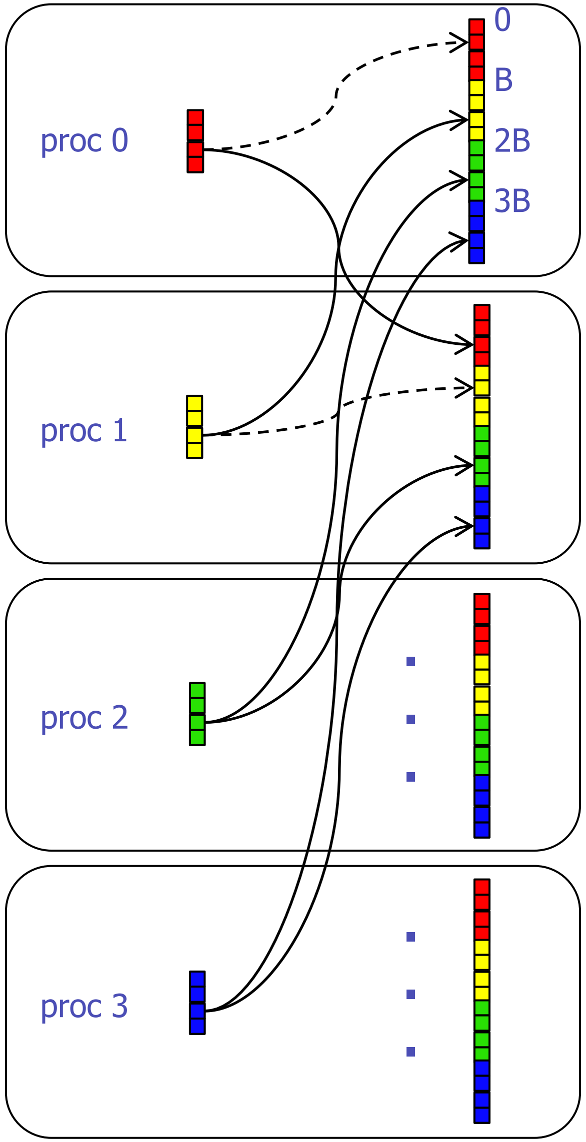


- Can assume this is array of x-positions.
- Actually need two more arrays for y and z coordinates (and perhaps three more for components of velocity).

Calculating Forces

- Each process updates positions and velocities of B stars.
- But when we come to calculate accelerations, need to know positions of every other star.
- Can use AllGather - which effectively combines a call to Gather with a call to Bcast - to get current position of every star to every process.
- Each process then does $N \times B$ individual force calculations.

AllGather Communication Pattern



AllReduce

- Finally, another equally useful "combined" operation is *AllReduce*, which behaves like a *Reduce* followed by a broadcast.
- For example, an alternative implementation of the stars simulation may see all star positions held *replicated* across all nodes.
- A node calculates the net force a block of these stars exert on on *all* other stars, then perform an *AllReduce* on the resulting array.
- All nodes redundantly perform the less demanding position/velocity updates.

Summary

- Finished our discussion of the MPI standard with an exploration of *collective communications*, which in MPJ are defined in the the Intracomm class.
- Standardizing these was a significant contribution of MPI, and they are a widely used part of the API.
- Discussion hasn't been exhaustive, and in fact the general idea of collective communication can be extended well beyond what is captured in MPI.

Further Reading

- MPJ Express API:

  http://mpj-express.org/docs/javadocs/index.html

- William Gropp, Ewing Lusk and Anthony Skjellum, *Using MPI*, 2nd Edition MIT Press, 1999.
  - Standard text on MPI, but examples are in C and Fortran.
  - Available as an electronic book through the library.