4. Distributed Memory, and MPI Basics

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth

Goals

- Set the background to *distributed memory programming* on clusters of computers.
- Basics of *MPI* – the parallel job.
- First example of message passing.

Limits of Shared Memory

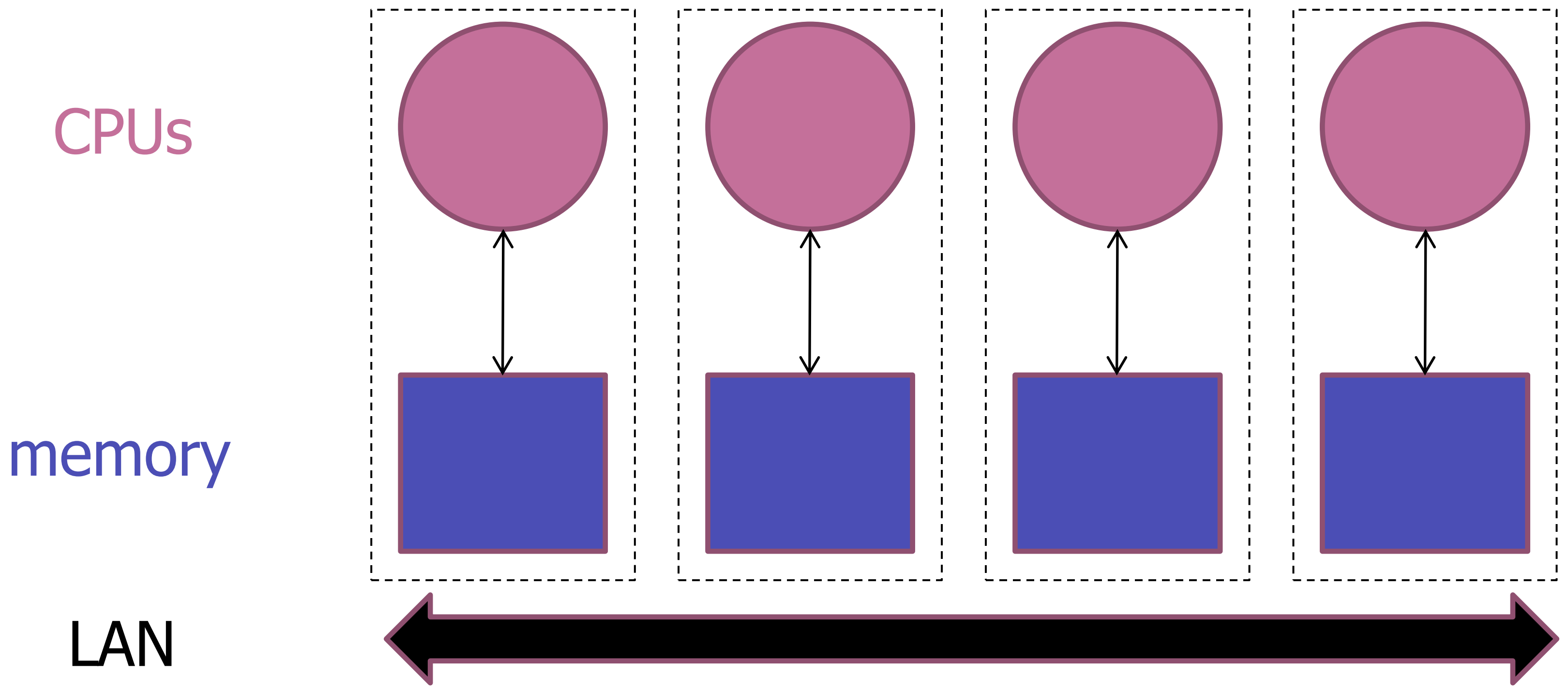
- Previous lecture concentrated on parallel programming with *shared memory*.
- Has become important in recent years with rise of commodity *multicore* processors.
- But widely accepted there are limits to the *scalability* of shared memory systems – how many cores can share a common memory.
- In the search for scalability, we are forced to consider *distributed memory systems*.

Distributed Memory

- *Distributed Memory* (DM) systems are not particularly mysterious or unfamiliar.
  - A group of ordinary workstations connected by a LAN is perhaps the most basic instantiation.
- In supercomputers this idea is pushed to extremes, with thousands of *nodes* (essentially the workstations) connected by a special purpose high speed network or *interconnect*
  - e.g. *Myrinet*, *Infiniband*, or custom interconnects.

A Simple Cluster

- Consider a group of four similar workstations:



- Total memory available is four times that of individual workstation, but *distributed* over group.

A Continuum

- Can identify a whole range of systems that *could* be used for *distributed memory* parallel computation:

Ad hoc grouping of workstations on TCP/IP LAN in lab

Clusters of commodity computers dedicated to parallel computing (e.g. *Beowulf*†)

Supercomputers with thousands of nodes on custom interconnect.

Increasing scale

† See for example http://en.wikipedia.org/wiki/Beowulf_cluster

From Threads to Processes

- From a software perspective, biggest change in programming these systems is that we move from *cooperating threads* - sharing access to a common memory - to *cooperating processes*, which each have their own *private memory*, but with no memory shared *between* processes.
- In our case these constraints are enforced by the physical hardware, but software considerations quite analogous to distinction between threads and processes within an individual *operating system*.

Programming

- *Programming* cooperating processes is generally *harder* than cooperating threads – often involves explicit *Inter-Process Communication* (IPC).
- For parallel processing at largest scales this seems hard to avoid.
- But quite similar programming techniques can be used over whole continuum of distributed memory systems introduced just now.

Message Passing

- In some sense the lowest common denominator approach, explicit Message Passing is *widely* used in DM parallel programs today.
- Many - probably most - parallel programs designed to run on large clusters and distributed memory supercomputers use one particular *API* (Application Programming Interface) for this.

The Message Passing Interface

- Some twenty-five years ago, *MPI version 1* was standardised by a group of manufacturers and academics called the *MPI Forum*.
  - Do *not* confuse *MPI* with *OpenMP*, which is a standard for *shared-memory* parallel programming we discussed last week†!
- *API* was defined for *C/C++* or *Fortran*, and still no *official* bindings beyond those two languages
  - though we will use an "unofficial" *Java* binding.

†To make things even more confusing, one of the popular open-source implementations of MPI is called *Open MPI*...

MPI BASICS

The MPI Programming Model

- MPI 1 assumes a fixed number (*P*, say) of processes participate in any given parallel program.
- This set of processes is called "*the world*"
  - Size of the world often but not always equal to number of physical nodes available†.
- MPI implements the *SPMD model* - each process runs *the same* program, but of course operates on its own local memory (data).

†MPI 2 introduced the option to create new processes during program execution -"growing" the world. Unclear how widely this feature is used.

SPMD Programming

- *Single Program Multiple Data* in which all participant processors (or processes) run the same program image, but operate on their local memory contents.
- A special case of the more general *MIMD* (*Multiple Instruction Multiple Data*) model, in which different participants may run different local programs.
  - e.g. a common paradigm in *pre-MPI* days was for one node to run a *host* program that coordinated I/O and controlled the other nodes, which ran a separate *worker* programs that did most of the computation.

Process Rank

- MPI processes in the world of a particular parallel program execution are distinguished by rank
- Just the MPI name for the numeric identifier of the processor.
- If the size of the world is *P*, the rank runs from 0 to *P* - 1†
- Directly comparable with me variable in our earlier parallel programming examples.

†Strictly this is rank relative to the world - ranks within *subgroups* of processes are also defined.

On Programming Examples

- We will generally use an *unofficial* Java API for MPI, as implemented in the software called *MPJ Express*.
- We will frequently refer to this Java API, *loosely*, as *MPJ*.
  - Strictly speaking, MPJ was another slightly different API†, but since we will be using *MPJ Express* in labs, we better stick to the that variant of the API.
- In any case, this API adheres as closely as practical to the official *C* standard for MPI.

†MPJ: MPI-like Message Passing for Java

MPJ Hello World

```
import mpi.* ;

public class HelloWorld {

    public static void main(String args[]) throws Exception {

        MPI.Init(args) ;

        int P    = MPI.COMM_WORLD.Size() ;
        int me = MPI.COMM_WORLD.Rank() ;

        System.out.println("Hi from <" + me + ">") ;

        MPI.Finalize() ;
     }
  }
```

MPJ Fundamentals

- Every MPJ program is a *Java application* starting with a main() method.
- This application is started in *P* logical processes, typically running on different nodes of the system.
- Every MPJ program *must* begin by calling:

```
MPI.Init(args) ;
```

where args is the parameter of main(), and finish by calling:

```
MPI.Finalize() ;
```

The World

- The object:

  MPI.COMM_WORLD

  is an *MPI Communicator* spanning the world – the universe of processes in which the program is running.
  - It is an instance of the class Comm, and in our binding many of the most useful methods in MPI belong to this class.
  - For now, just accept that this is a Java object – and that most of the methods we see initially are called on this object.

Number of Processes and Process Rank

- In MPI, the number of processes the program runs in is *not* defined by the program itself.
- Instead it is defined by the command used to run the parallel program.
- The program can interrogate the number of processes by calling:

```
MPI.COMM_WORLD.Size()
```

- Each running process can find its "position" in the set of processes by calling:

```
MPI.COMM_WORLD.Rank()
```

Running MPJ Hello World

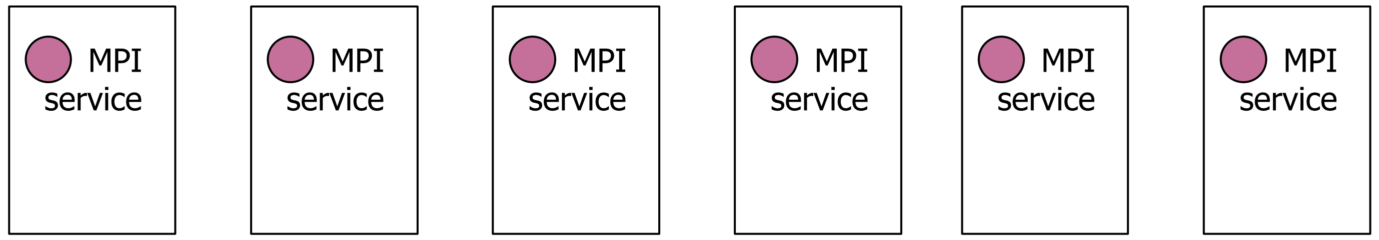
- At the command prompt on client computer, might run:

```
> mpjrun -np 4 HelloWorld
 Hi from <1>
 Hi from <3>
 Hi from <2>
 Hi from <0>
```

- Of course there is no particular order to the output from the 4 processes.

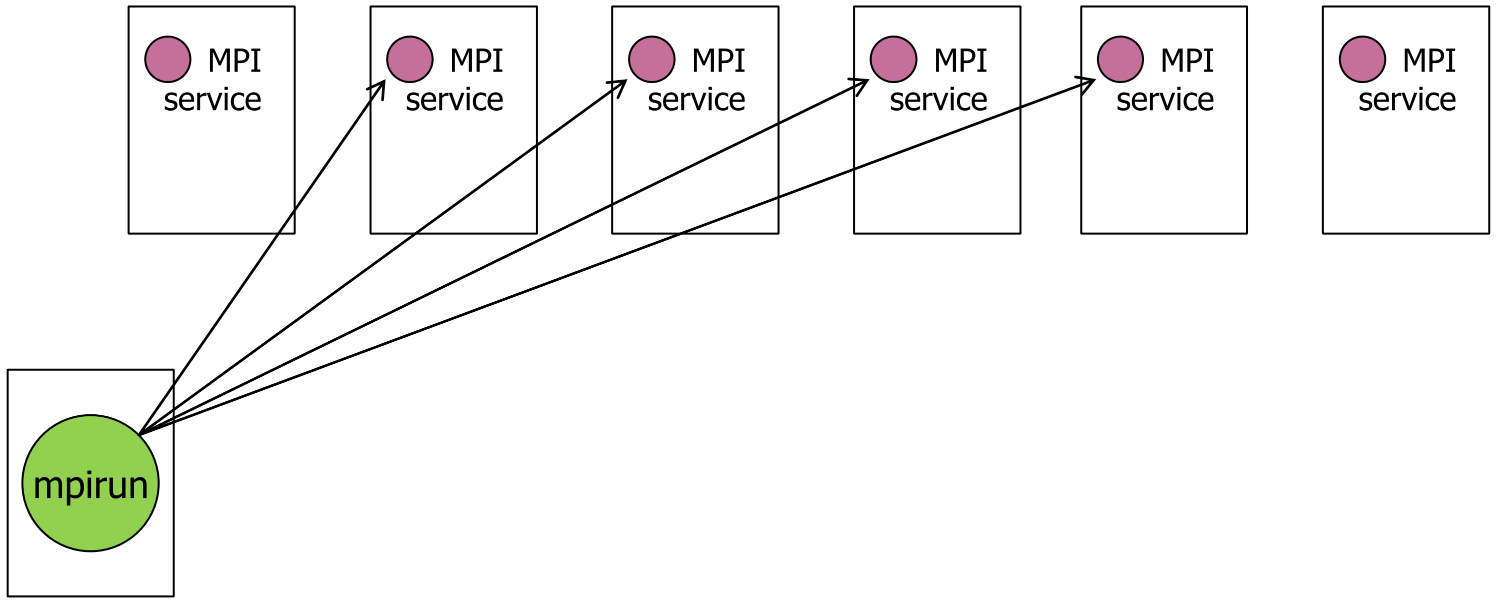
Scenario for Running an MPI Program

- Pool of available host computers, each running an *MPI service* or *MPI daemon*.



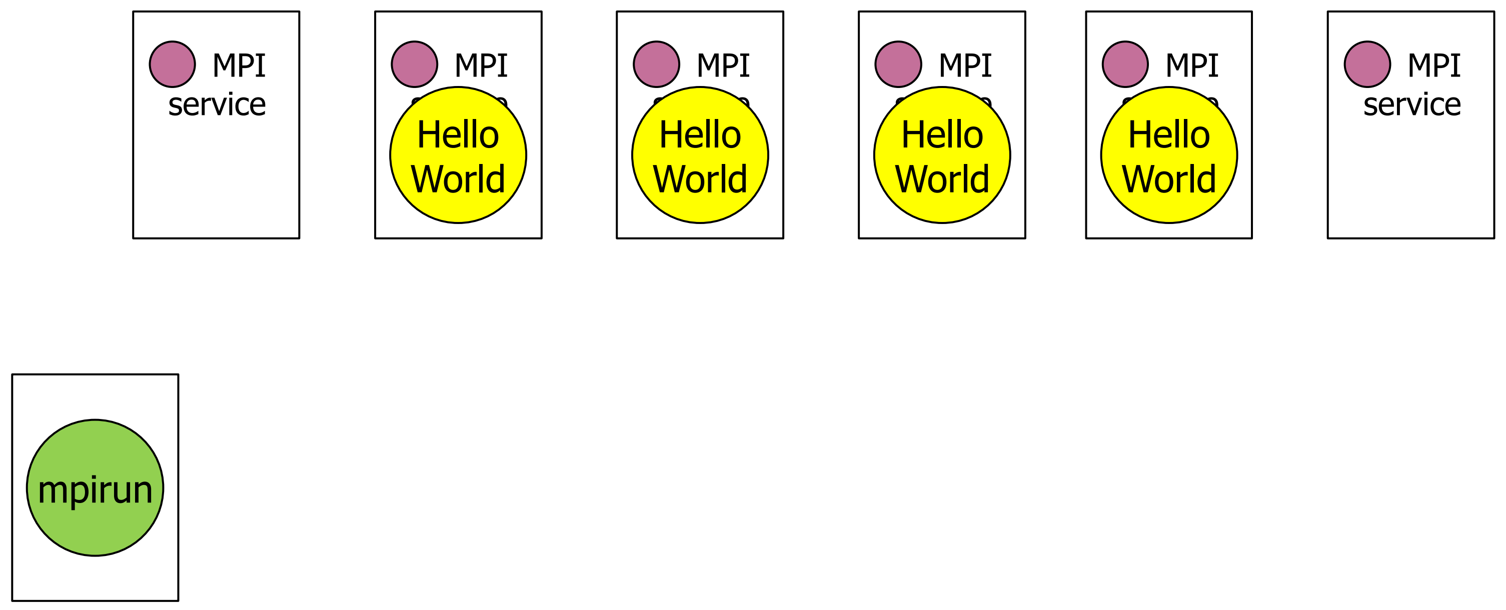
Scenario for Running an MPI Program

- Client program, e.g. mpirun, connects to *P* daemons and asks them to start processes.



Scenario for Running an MPI Program

- *Hello World* process (say) starts on *P* hosts.



Handling Print Statements

- Commonly, MPI daemons redirect any *standard output* of the *P* distributed processes (from print statements, etc) back to the client program - mpirun in our example
  - It is displayed on the console of the client process.
  - Useful for debugging, etc.
- Other I/O such as file input or output is *not* handled automatically – program has to figure out exactly where it wants it to go!

MESSAGE PASSING

Simple Message Passing

- The main point of MPI is to allow our P processes to *communicate* in order to collaborate on tasks.
- The simplest method for *sending* a message in MPJ is:

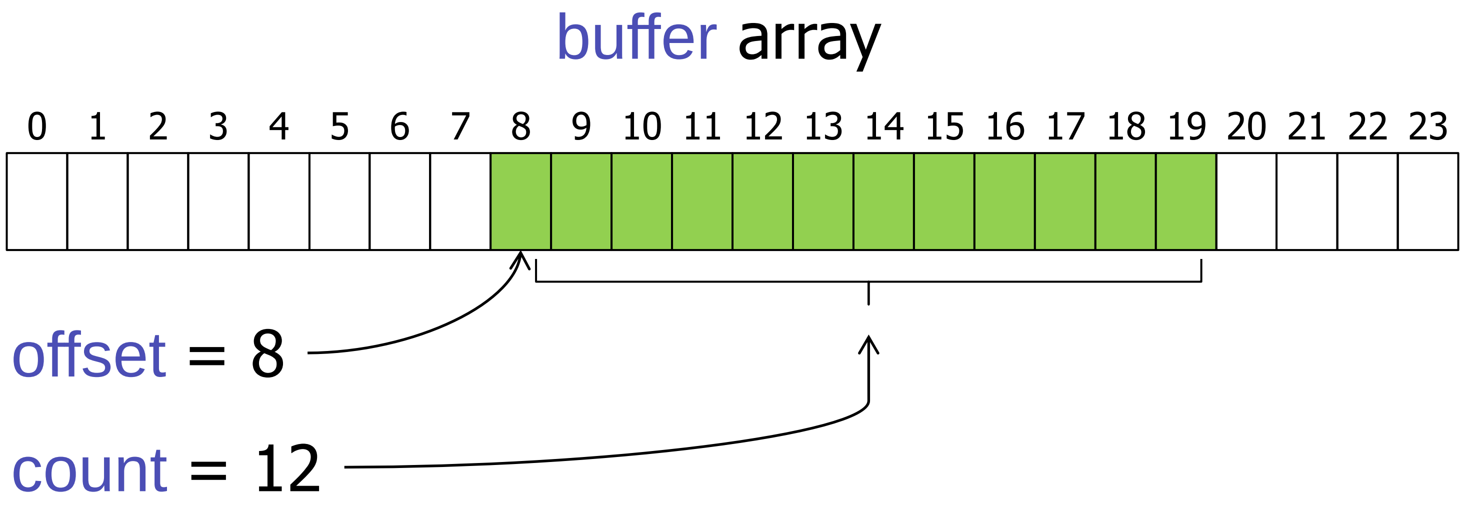
  Send(buffer, offset, count, type, dest, tag)

- Unfortunately, the argument lists of operations in MPI tend to be long...

Message Buffer

- The *message buffer* is just an array in the user's program that contains the data to be sent to another program.
- The buffer argument of **Send()** is therefore a Java array containing this information.
- The offset and count arguments select the elements of buffer that are actually going to be sent in the message.
- The type argument defines the type of the elements of buffer. It is slightly redundant in Java – just follow the examples for now.

Buffer Example



- Elements of buffer actually sent are in green.
- Note offset is often 0.
- count may take value 1 to send a single element.

Destination and Tag

- The dest argument of **Send()** is just the numerical id – i.e. the *rank* - of the process to which the message is to be sent.
- The tag argument is a user-defined code for identifying the purpose of the message – for the time being we will just set it to 0.

Receiving a Message

- The simplest method for *receiving* a message in MPJ is:

  Recv(buffer, offset, count, type, src, tag)

- Arguments are almost the same as **Send**().
- buffer is now an array into which the contents of the incoming message will be placed; offset says where in the array, and count is the maximum message size that can be accepted.
- src is now the rank of the process we expect the message to come *from*.

An Example

- The following slide gives a sequential algorithm for calculating π, based on the following formula from Calculus:

$$\pi = \int_0^1 \frac{4}{1 + x^2}\, dx$$

  applying the "*rectangle rule*" for numerical integration (don't worry about the math!)
- N is the number of steps used for the numerical approximation.
- For clarity, we only give the central part of the algorithm.

Sequential π Calculation

```
double step = 1.0 / (double) N;
double sum = 0.0;
for(int i = 0 ; i < N ; i++) {
    double x = (i + 0.5) * step ;
    sum += 4.0 / (1.0 + x * x);
}
double pi = step * sum ;
```

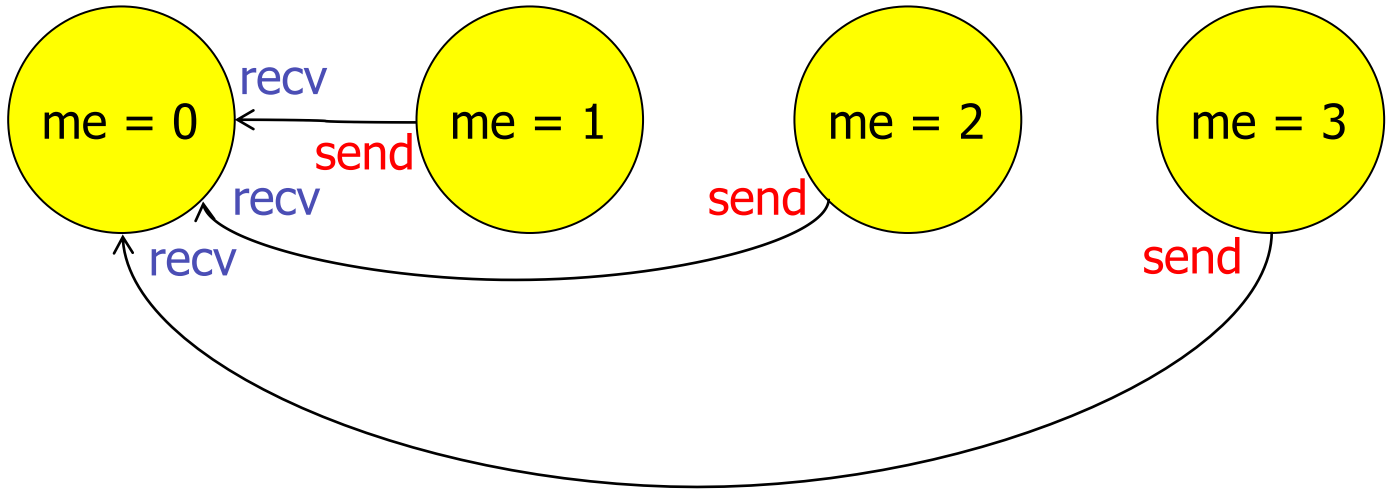
MPJ Parallel π Calculation (Part 1)

```
int me = MPI.COMM_WORLD.Rank() ;
int P    = MPI.COMM_WORLD.Size() ;

int b = N / P ;
int begin = me * b;
int end = begin + b ;

double step = 1.0 / (double) N ;
double sum = 0.0 ;
for(int i = begin ; i < end ; i++) {
      double x = (i + 0.5) * step ;
      sum += 4.0 / (1.0 + x * x);
}
```

Comments

- This is the easy part, and it is very much analogous to block-wise decompositions we have seen before in shared memory examples
- The number of processes, *P*, and the process identity *me* are now obtained by the MPJ inquiry functions Size() and Rank().

Communication Pattern Needed



MPJ Parallel π Calculation (Part 2)

```
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 ;
```

Comments

- This is rather harder to understand – there is a lot to digest here.
- Once absorbed, you should have a clearer idea of what SPMD programming with MPI is about!
    - Processes with me > 0 are sending a message containing their contribution to the sum to process 0.
    - Process 0 receives a message from processes 1 to *P*-1, and accumulates the value it receives into its contribution to the sum.
- Process 0 now holds the final result.

Summary

- Next week: *Communication in MPI*.