# High-Performance Computing: MPI

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

2015–16

# MPI, the **M**essage **P**assing **I**nterface

MPI is was standardized in 1994, and revised in 1997 and 2012

It is *the* standard way for performing high-performance computing on clusters, networks of workstations, the Grid, and so on

It is based around the explicit passing of messages from one machine to another, so that they can work in parallel on a problem

MPI is an application programming interface — a library that you link a program with

MPI can be used from C, C++, Fortran, Java, and Python

# MPI Goals

Provide efficient communication, avoiding memory-to-memory copying and allowing overlap of computation and communication

Support heterogeneous computing environments

Provide a reliable interface, so that the programmer does not have to handle communication failures

Be implementable on a variety of vendor's platforms (yes, it can even run on Windows)

Be thread-safe

# Features

Point-to-point communication between processes on processors

Collective operations

Process groups

Communication domains

Process topologies

Environment enquiry and management

Profiling interface

Altogether, there are about 125 MPI routines — but you need only six to get going

# The essential calls

`MPI_Init` — initialize MPI

`MPI_Comm_size` — find how many processes are used

`MPI_Comm_rank` — determine which process this is

`MPI_Send` — send a message

`MPI_Recv` — receive a message

`MPI_Finalize` — finish using MPI

Listing **mpi-01.c** shows a minimal MPI program

All MPI programs must invoke `MPI_Init()` first and `MPI_Finalize()` last

Each statement executes independently in each process, so this is an MIMD system

`MPI_COMM_WORLD` designates all processes in the MPI "job"

`printf` is not part of MPI and so the output from processes may be interleaved

MPI does not define how programs are compiled, though MPICH provides `mpicc` for C and `mpicxx` for C++

However, it *does* define a standard way in which MPI programs are run

# Running an MPI program

To run a program with 16 processes:

```
mpiexec -n 16 myprog
```

If you would like 12 processes but will make do with a smaller number instead:

```
mpiexec -n 12 -soft 1:12 myprog
```

To run on a machine called `wombat`:

```
mpiexec -n 12 -soft 1:12 -host wombat myprog
```

To run on a specific architecture and pass `arg1` to `myprog`:

```
mpiexec -n 12 -soft 1:12 -arch sparc:solaris myprog -arg1
```

To run 5 processes of `myprog` and 10 of `myprog2`:

```
mpiexec -n 5 myprog1 : -n 10 myprog2
```

As this is a bit of a fingerful, you can also use a configuration file:

```
mpiexec -configfile conf.txt
```

containing

```
-n 5 myprog1
-n10 myprog2
```

# Doing different things

Listing **mpi-02.c** shows a program that does something different in one process compared to all the others

The "worker" processes each generate a message and return it to the "controller" process, which we have chosen to be process 0

When this program is run, it will output lines, each of which contains a "hello" message from each worker — but the order in which the lines are output may vary each time the program is run

We can clearly extend this approach so that something different is performed in each program of the larger system

# Finding out about the other processes

As we have seen, `MPI_Comm_rank` returns the rank of the calling process as its second argument:

```
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

- The first argument is a *communicator*, essentially a set of processes that can inter-communicate
- `MPI_COMM_WORLD` is a communicator that encompasses all processes

Similarly, `MPI_Comm_size` returns the total number of processes:

```
int MPI_Comm_size (MPI_Comm comm, int *size)
```

# Sending and receiving messages

The most basic message-passing functions in MPI are provided by

- `MPI_Send` sends a message to a specific process
- `MPI_Recv` receives a message from a specific process

In addition to the message itself (the 'payload'), MPI adds some information (the 'envelope'):

- the rank of the receiver
- the rank of the sender
- a communicator
- a tag

The `src` argument in `MPI_Recv` identifies which process sent the received message

The `tag` is a user-specified `int` that can be used to distinguish messages received from a single process:

- Suppose process 1 needs to send two values to process 0, each of which is a single integer but they have different meanings in process 0
- Using different `tag` values gives process 0 a way of distinguishing which value is which
- The values 0–32767 are guaranteed to be available for `tag`, though most implementations allow many more than this

# Why tags *and* communicators?

As tags provide a fairly generic way of identifying messages, why are there also communicators?

The answer is that communicators make it easier to write libraries (*e.g.*, to solve sets of equations in parallel) without them inadvertently having their messages received by other processes

Note that there are also two 'wildcards' that can be used in `MPI_Recv`:

- `MPI_ANY_TAG` to catch an incoming message with any tag
- `MPI_ANY_SOURCE` to catch an incoming message from any source in the communicator

```
int MPI_Send (void *message, int count, MPI_Datatype type,
    int dest, int tag, MPI_Comm comm)

int MPI_Recv (void *message, int count, MPI_Datatype type,
    int src, int tag, MPI_Comm comm, MPI_Status *status)
```

Together, `count` and `type` determine the size of the chunk of memory in `message`

Most of the standard C datatypes are supported, and `MPI_BYTE` can be used to tell the system to perform no conversion of the data in `message`

The receive buffer has to be large enough to hold an entire incoming message

| MPI datatype | C datatype |
|---|---|
| `MPI_CHAR` | `signed char` |
| `MPI_SHORT` | `signed short int` |
| `MPI_INT` | `signed int` |
| `MPI_LONG` | `signed long int` |
| `MPI_FLOAT` | `float` |
| `MPI_DOUBLE` | `double` |
| `MPI_LONG_DOUBLE` | `long double` |
| `MPI_BYTE` | |
| `MPI_PACKED` | |

There are also unsigned versions of all the integer types, such as `MPI_UNSIGNED_CHAR` for `unsigned char`

The `status` return from `MPI_Recv` is actually a structure with two fields, and these are used to receive the actual error

```
int rectag, recfrom, recval;
MPI_Status status;
...
MPI_Recv (..., &status);
revtag = status.MPI_TAG
recfrom = status.MPI_SOURCE
MPI_Get_count (&status, datatype, &recval);
```

# Communication modes

Implicit in our discussion to date is that the execution of a program that performs `MPI_Send` is suspended until its entire message has been buffered or sent

Similarly, a program that calls `MPI_Recv` will wait until the entire incoming message has been received

These are characteristics of *blocking* communication. MPI also provides routines for *non-blocking* communication (both send and receive), and determining the completion status of a non-blocking operation

If used with care, non-blocking can often result in faster programs

| mode | blocking routine | non-blocking routine |
|---|---|---|
| standard send | `MPI_Send` | `MPI_Isend` |
| receive | `MPI_Recv` | `MPI_Irecv` |
| synchronous send | `MPI_Ssend` | `MPI_Issend` |
| buffered send | `MPI_Bsend` | `MPI_Ibsend` |
| ready send | `MPI_Rsend` | `MPI_Irsend` |

*Synchronous send* completes only when the receive has completed

*Buffered send* creates a buffer, copies the data and returns control, so it always completes, irrespective of the receiver

*Ready send* similar to buffered send but always completes, irrespective of whether the receive has completed

```
int MPI_Isend (void *message, int count, MPI_Datatype type,
    int dest, int tag, MPI_Comm comm, MPI_Request *req)

int MPI_Irecv (void *message, int count, MPI_Datatype type,
    int src, int tag, MPI_Comm comm, MPI_Request *req)

int MPI_Wait (MPI_Request *req, MPI_Status *status)

int MPI_Test (MPI_Request *req, int *flag,
    MPI_Status *status)
```

`MPI_Wait` waits until a communication has completed

`MPI_Test` determines whether a communication has completed — `flag` is set to `true` if it has

# Example: Calculating $\pi$

This is one of the standard examples of how to program using MPI

We make use of the relationship that

$$\int_0^1 \mathrm{d}x \frac{4}{1+x^2} = \pi$$

The plan is to split the curve into a large number of small rectangles and add them up — this is a form of *numerical integration*:

$$\pi \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) \quad \text{where} \quad x_i = \frac{i - \frac{1}{2}}{N}, i = 1, 2, \ldots N$$

On my Mac, using $10^{10}$ bins, a C program to calculate this takes about 11 seconds — see **mpi-03.c**

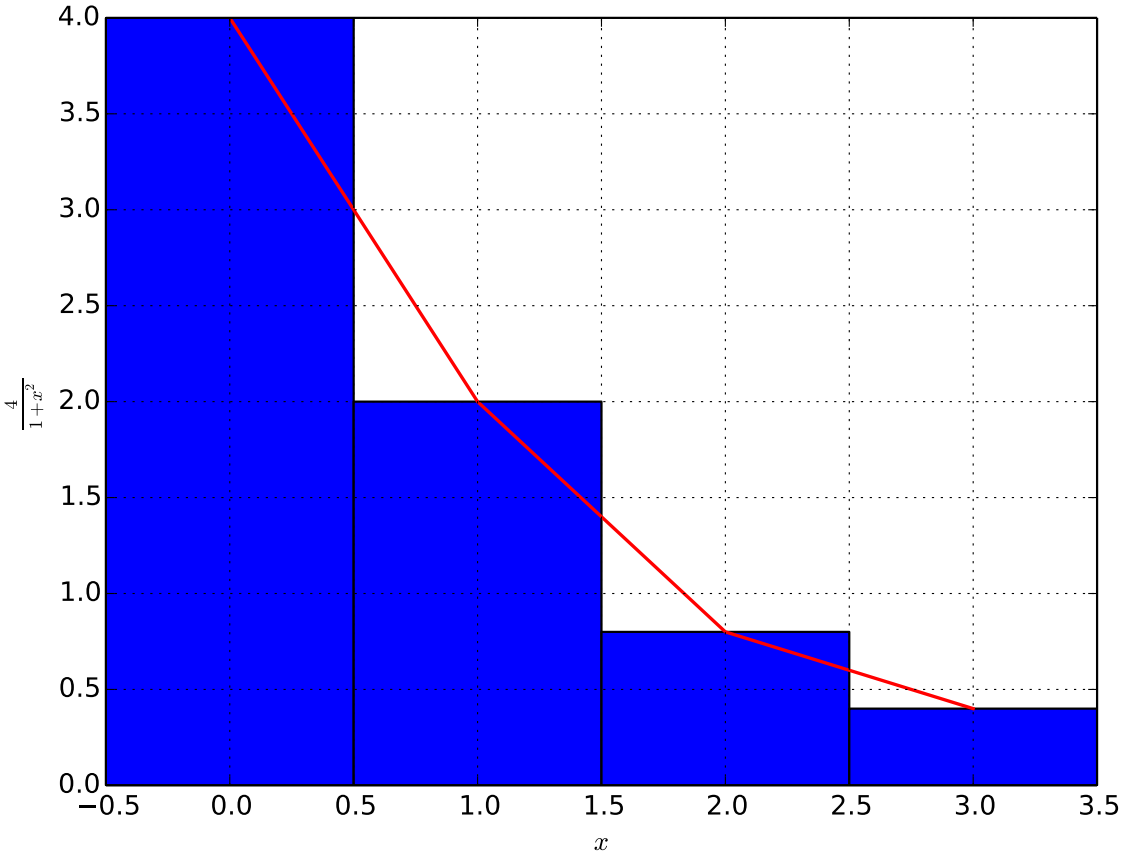


Figure 1:graph of $\frac{4}{1+x^2}$