

# Message Passing Interface (MPI)

## What the course is:

- An introduction to parallel systems and their implementation using MPI
- A presentation of all the basic functions and types you are likely to need in MPI
- A collection of examples

## What the course is not:

- A complete list of all MPI functions and the context in which to use them
- A guarantee that you will never be faced with a problem requiring those functions
- An alternative to a book on MPI

# Message Passing Interface (MPI)

## How the course will work:

- Five one-hour lectures
- Slides available on the web (if possible in advance of the lecture)  
<http://computing.dcu.ie/~dperrin/teaching.html#MPI>
- Course outline:
  - Introduction
  - When is a parallel implementation useful?
  - The basics of MPI
  - Some simple problems
  - More advanced functions of MPI
  - A few more examples

# Message Passing Interface (MPI)

## Want more?

Tutorials: A few MPI tutorials are listed at: <http://computing.dcu.ie/~dperrin/teaching.html#MPI>

Source code: Commented code for all examples will also be available at the same address.

Books: - “Using MPI – Parallel programming with the Message-Passing Interface”,

Gropp, Lusk and Skjellum, ISBN: 0-262-57134-X

<http://www.powells.com/biblio?isbn=026257134x>

- “Using MPI-2 – Advanced features of the Message-Passing Interface”,

Gropp, Lusk and Thakur, ISBN: 0-262-057133-1

<http://www.powells.com/biblio?isbn=0262571331>

# Message Passing Interface (MPI)

## Why/when would you use a parallel approach?

- Large problems
- Problems suitable for parallelisation, i.e. you know what speed-up to expect  
=> you need to be able to recognise them

## Three types of problems are suitable:

- Parallel Problems
- Regular and Synchronous Problems
- Irregular and/or Asynchronous Problems

# Message Passing Interface (MPI)

## Parallel problems:

- The problem can be broken down into subparts
- Each subpart is independent of the others
- No communication is required, except to split up the problem and combine the final results
- Linear speed-up can be expected

Ex: Monte-Carlo simulations

# Message Passing Interface (MPI)

## Regular and Synchronous Problems:

- Same instruction set (regular algorithm) applied to all data
- Synchronous communication (or close to): each processor finishes its task at the same time
- Local (neighbour to neighbour) and collective (combine final results) communication
- Speed-up based on the computation to communication ratio
  - => if it is large, you can expect good speed-up for local communications and reasonable speed-up for non-local communications

Ex: Fast Fourier transforms (synchronous), matrix-vector products, sorting (loosely synchron.)

# Message Passing Interface (MPI)

## Irregular and/or Asynchronous Problems:

- Irregular algorithm which cannot be implemented efficiently except with message passing and high communication overhead
- Communication is usually asynchronous and requires careful coding and load balancing
- Often dynamic repartitioning of data between processors is required
- Speed-up is difficult to predict; if the problem can be split up into regular and irregular parts, this makes things easier

Ex: Melting ice problem (or any moving boundary simulation)

# Message Passing Interface (MPI)

## A first example: matrix-vector product

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

$$\text{with } \begin{cases} c_1 = a_{11} \times b_1 + a_{12} \times b_2 + a_{13} \times b_3 + a_{14} \times b_4 \\ c_2 = a_{21} \times b_1 + a_{22} \times b_2 + a_{23} \times b_3 + a_{24} \times b_4 \\ c_3 = a_{31} \times b_1 + a_{32} \times b_2 + a_{33} \times b_3 + a_{34} \times b_4 \\ c_4 = a_{41} \times b_1 + a_{42} \times b_2 + a_{43} \times b_3 + a_{44} \times b_4 \end{cases}$$

# Message Passing Interface (MPI)

## A parallel approach:

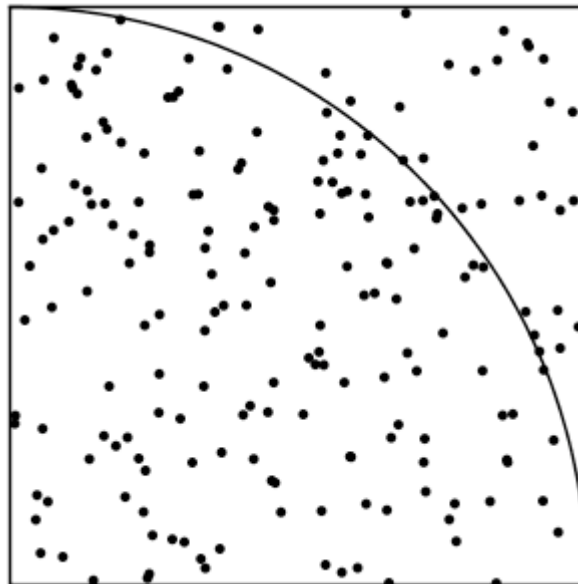
- Each element of vector  $c$  depends on vector  $b$  and only one line of matrix  $A$
- Each element of vector  $c$  can be calculated independently from the others
- Communication is only needed to split up the problem and combine the final results

=> a linear speed-up can be expected, for large matrices

# Message Passing Interface (MPI)

## Another example: Monte-Carlo calculation of Pi

- $\pi = 3.14159\dots$
- $\pi = \text{area of a circle of radius } 1$
- $\pi/4 \approx \text{fraction of the points within the circle quadrant}$
- The more points we have, the more accurate the value for  $\pi$  is



# Message Passing Interface (MPI)

## A parallel approach:

- Each point is randomly placed within the square
- The position of each point is independent of the position of the others
- We can split up the problem by letting each node randomly place a given number of points
- Communication is only needed to specify the number of points and combine final results

=> a linear speed-up can be expected, allowing for a larger number of points and therefore a greater accuracy in the estimation of  $\pi$ .

# Message Passing Interface (MPI)

## Something more “real”: chess software

- After each move, the chess software must find the best move within a set
  - This set is large, but finite
  - Each move from this set can be evaluated independently, and the set can be partitioned
  - Communication is only needed to split up the problem and combine the final results
- => A linear speed-up can be expected
- => This means that, in a reasonable time, moves can be studied more thoroughly
- => This depth of evaluation is what makes the software more competitive

# Message Passing Interface (MPI)

## Some background on MPI:

- Who?
  - MPI forum (made up of Industry, Academia and Govt.)
  - They established a standardised Message-Passing Interface (MPI-1) in 1994
  
- Why?
  - Intended as an interface to both C and FORTRAN
  - C++ bindings in MPI-2. Some Java bindings exist but are not standard yet
  - Aim was to provide a specification which can be implemented on any parallel computer or cluster; hence portability of code was a big aim

# Message Passing Interface (MPI)

## Advantages of MPI:

- Portable, hence protection of software investment
- A standard, agreed by everybody
- Designed using optimal features of existing message-passing libraries
- “Kitchen-sink” functionality, very rich environment (129 functions)
- Implementations for Fortran, C and C++ are freely downloadable

# Message Passing Interface (MPI)

## Disadvantages of MPI:

- “Kitchen-sink” functionality, very rich environment (129 functions). Hard to learn all (but this not necessary: a bare dozen are needed in most cases)
- Implementations on shared-memory machines is often quite poor, and does not suit the programming model
- Has rivals in other message-passing libraries (e.g. PVM)

# Message Passing Interface (MPI)

## Preliminaries:

MPI provides support for:

- Point-to-point & collective (i.e. group) communications
- Inquiry routines to query the environment (how many nodes are there, which node number am I, etc.)
- Constants and data-types

We will start with the basics: initialising MPI, and using point-to-point communication.