# Message Passing Interface
# a brief introduction

**University of Birmingham**

30th January 2017

**Ning Li -** Senior Technical Consultant

**Sally Bridgwater** – HPC Application Analyst

nag®

Experts in numerical software and
High Performance Computing

## What is MPI

- MPI is short for **M**essage-**P**assing **I**nterface.
- MPI is designed by a consortium of organisations as a standard for writing message-passing programs.
- We work with MPI libraries, which are implementations of the MPI specification.
- There are many versions of the MPI standard
  - MPI-3.1 is the latest standard. Several implementations are available, but not yet universally supported.
  - Most mature implementations are against the MPI-2.1 or 2.2 standard.

## Why is MPI the Library of Choice?

- MPI is the de-facto standard of parallel programming for distributed memory systems.
- Portable code
  - Implementations exist for most parallel platforms.
  - Free, downloadable implementations available.
- Optimal performance
  - Considerable effort has been put into optimising the performance of the library and tuning it to specific hardware platforms and interconnects.
  - Such development is ongoing.
- The standard itself is also continually being refined and updated.

## MPI Implementations

- Two widely used open-source implementations
  - Use them for studying and development work on your PC/laptop.
  - OpenMPI – http://www.open-mpi.org
  - MPICH – http://www.mpich.org
- Popular vendor implementations
  - Often high-performance on the platforms they are designed for.
  - Popular implementations: Intel MPI, Platform (IBM), Cray MPT, etc.
- Tips
  - Because MPI is a standard, your code should work with any implementation.
  - In practice, for difficult situations, such as debugging or performance analysis, trying a different MPI library is often a good idea.

## MPI Language Bindings

- There are official MPI bindings for C and Fortran.
- A C++ binding was introduced at MPI-2.0, but deprecated at MPI-2.2 and removed at MPI-3.0.
  - C++ programmers should use the C bindings.
- A new Fortran 2008 binding has been added at MPI-3.0, although not supported universally.
- Third-party supports available for other languages
  - Python binding via mpi4py or similar extensions
  - R bindings of MPI via Rmpi
  - Etc.

## MPI Code Essentials

- A typical MPI code will have all of the following essentials elements:
  - Including the appropriate header file or module.
  - Initialising the MPI environment.
  - Getting each MPI process to find out the total number of processes, known as the **size** of the global communicator.
  - Getting each MPI process to find out its own unique ID, known as its **rank.**
  - Implementing some useful algorithms (normally decompose your problem based on **size** and **rank**, allowing each MPI process to handle a portion of the global workload).
  - Shutting down the MPI environment.
- We will briefly go through all these steps. After that you are able to create your first MPI program.

## The MPI Header File / Module

- To make the MPI defined constants and functions available to user code.

```
/* In C or C++, include the header file. */
#include <mpi.h>

! In Fortran, always use an MPI module if one is
! available on your system. An MPI-2.0 (or later)
! compliant implementation should provide one.
USE MPI ! Or USE MPI_f08 for the Fortran 2008 binding

! Otherwise, include the Fortran header file.
include 'mpif.h'
```

## Initialising the MPI Environment

- **MPI_Init** initialises the MPI environment.
  - All MPI codes must contain exactly one call to an initialisation routine.
  - Multi-threaded code may alternatively call **MPI_Init_thread**.
  - Calling most MPI routines before initialisation is a mistake.

```
/* C and C++ startup routine */
int MPI_Init(int *argc, char ***argv);
/* These are pointers to the arguments to main. It is
   permitted to pass NULL for both arguments. */

! Fortran startup
SUBROUTINE MPI_INIT(IERROR)
INTEGER :: IERROR
```

## Finalizing the MPI Environment

- Each process must call **MPI_Finalize** before it exits.
  - The user needs to ensure that all pending communication has completed before calling it.
  - This routine is responsible for shutting down the MPI environment and claim back system resourced used by it.

```
/* C and C++ shut-down routine */
int MPI_Finalize();
```

```
! Fortran shut-down
SUBROUTINE MPI_FINALIZE(IERROR)
INTEGER :: IERROR
```

## Definition of Rank and Size

- **size** is the total number of MPI processes.
  - This number is normally specified at runtime.
- **rank** is a unique integer associated with each process, where 0 <= **rank** < **size**.
- With **size** and **rank**, it is the programmer's responsibility to find a way to decompose the problem so that different processes perform different tasks or work on different data.
- Strictly speaking, **rank** and **size** should be associated within a group of processes called a *communicator*. For this talk, we work with **MPI_COMM_WORLD** only.

## Finding out the Size

- The function **MPI_Comm_size** reports the size of the group of processes associated with the specified communicator.

```
/* C and C++ */
int MPI_Comm_size(MPI_Comm comm, int *size);

! Fortran
SUBROUTINE MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER :: COMM, SIZE, IERROR
```

## Finding out the Rank

- The function **MPI_Comm_rank** finds the rank of a process within the group of processes associated with the specified communicator.

```
/* C and C++ */
int MPI_Comm_rank(MPI_Comm comm, int *rank);

! Fortran
SUBROUTINE MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER :: COMM, RANK, IERROR
```

## An MPI C Template

```
#include <mpi.h>
int main(int argc, char ** argv) {
   int size, rank;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   /* the body of the code goes here */

   MPI_Finalize();
}
```

## An MPI Fortran Template

```
PROGRAM basic_MPI_template
  USE MPI
  IMPLICIT NONE
  INTEGER :: ierror, rank, size
  CALL MPI_INIT(ierror)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

  ! the body of the code goes here

  CALL MPI_FINALIZE(ierror)
END PROGRAM basic_MPI_template
```

## An MPI Python Example

```
#!/usr/bin/env python

from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.rank
print "Hello world from rank", rank
```

## Compiling and Linking MPI Programs

- Use the MPI compiler wrappers
  - Normally ***mpif90*** for Fortran, ***mpicc*** for C and ***mpiCC*** for C++
  - Note that MPI is implemented as a standard library
  - When building user code, compiler needs ext locate external libraries
    - Such information is automatically supplied by the MPI compiler wrappers
- For example, with GNU compiler

```
gcc test.c –o test # to compile a serial code
mpicc mpitest.c –o mpitest # to compile an MPI code
mpicc –show mpitest.c –o mpitest
gcc mpitest.c -o mpitest -I/usr/include/openmpi-x86_64 -
  pthread -Wl,-rpath -Wl,/usr/lib64/openmpi/lib -Wl,--
  enable-new-dtags -L/usr/lib64/openmpi/lib -lmpi
```

## Running MPI Programs

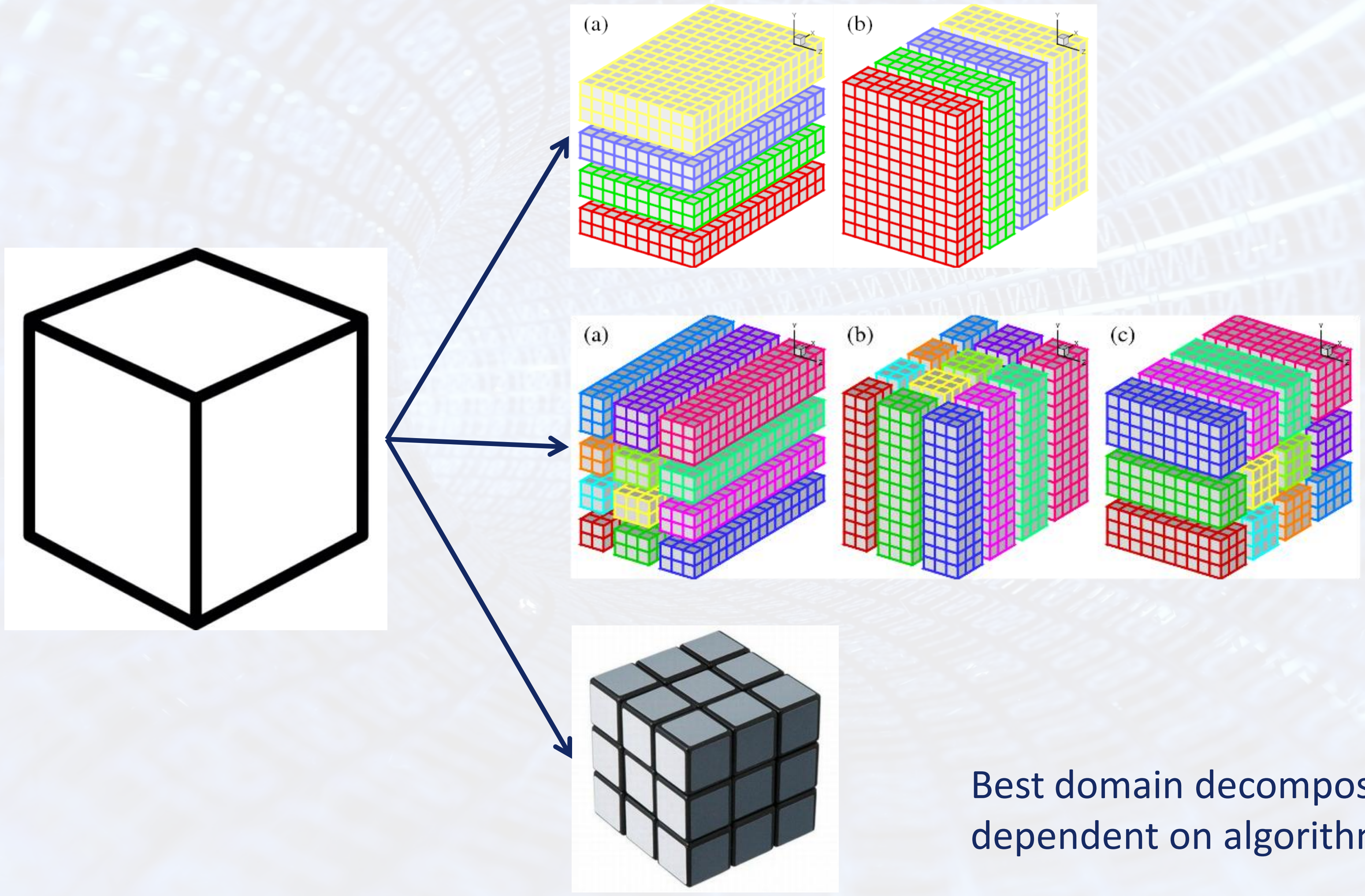
- Interactive runs
  - Mainly during development
  - Run job interactively from command line, e.g.

```
mpirun -np 64 ./name_of_executable -command_line_arguments
```

- Batch runs
  - Queue jobs on HPC systems
  - Major job scheduling systems
    - PBS, LSF, Slurm
  - Use a job script to
    - Reserve system resources
    - Set up runtime environment
  - Utility programs to
    - Submit job, check job queue …

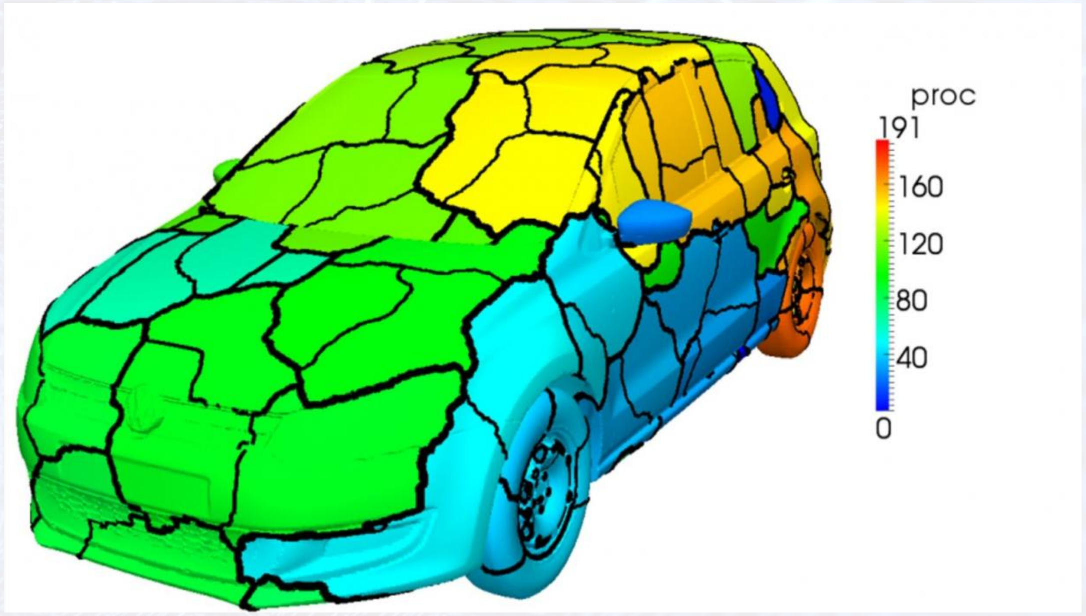
```
#!/bin/bash -login
# A sample job script

#PBS -N name_of_the_MPI_Job
#PBS -l select=144
#PBS -l walltime=00:20:00
#PBS -l place=excl
#PBS -A i213

cd $PBS_O_WORKDIR # job submission directory

# Load necessary environment
module load mpt
module load intel-compilers-16

# To actually launch the parallel job
mpirun -n 72 -ppn 36 ./hello.exe
```

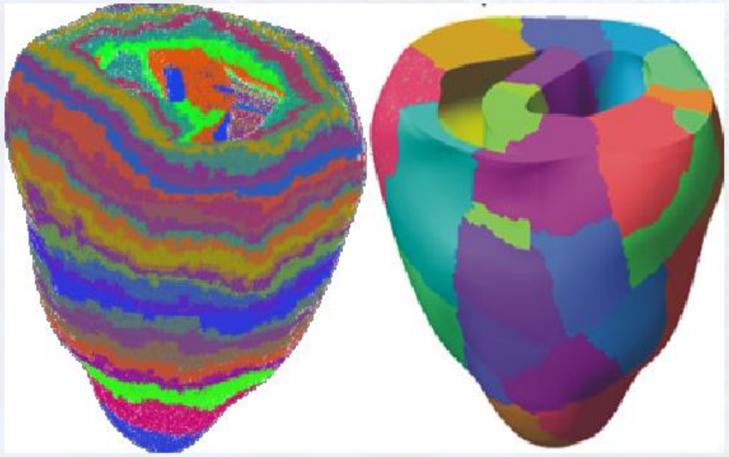
# Domain Decomposition Example – Structured Mesh



Best domain decomposition is dependent on algorithm.

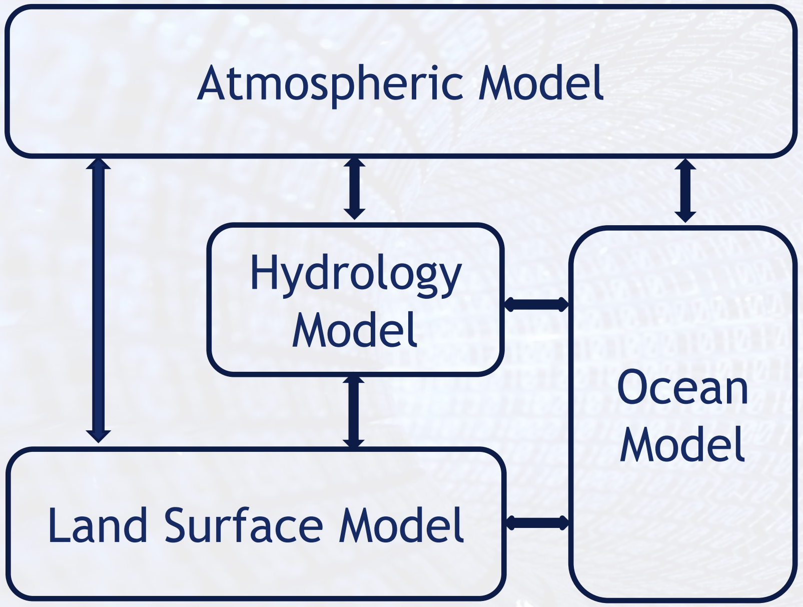
## Domain Decomposition Example – Unstructured Mesh





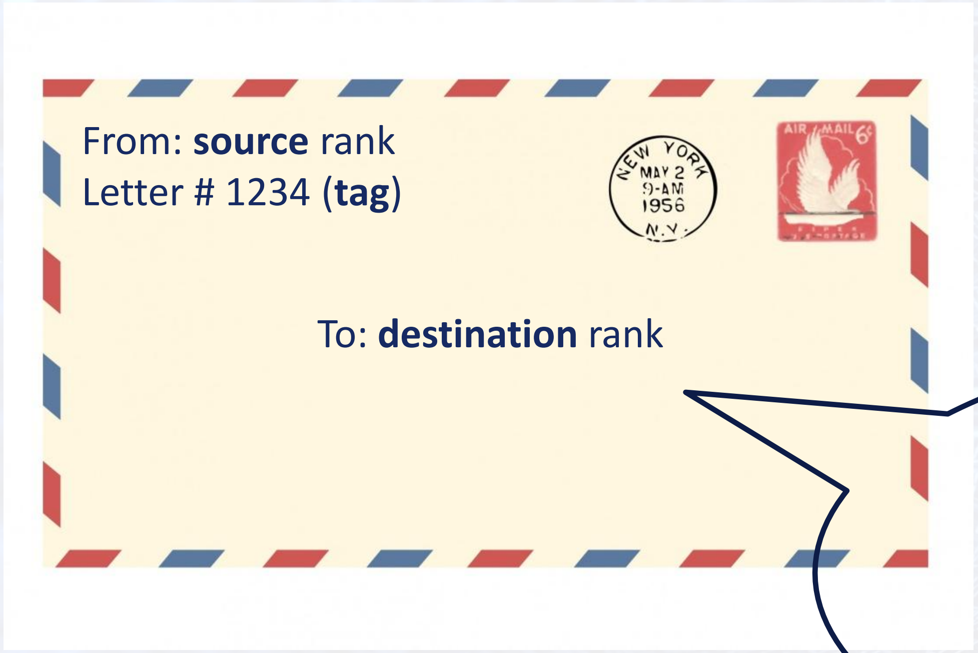
3rd party partitioning software often required

## Functional Decomposition Example



A subset of processes responsible for each model

## Messages in MPI



The message has:
**A type** - integer, real, etc.
**A count**
**A location** - memory address
Both **source** and **destination** ranks need to provide these

Please send **count** number data of this **type**, located at memory address a on the **source** rank, to the **destination** rank and write to memory address b there.

## Sending Data with MPI_Send

- The API maps to the concept very well, except the added **communicator** argument

```
/* C and C++ */
int MPI_Send(void *buf, int count,
  MPI_Datatype datatype, int dest, int tag,
  MPI_Comm comm);

! Fortran
SUBROUTINE MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG,
  COMM, IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

## Receiving Data with MPI_Recv

- The API maps to the concept very well, except the added **communicator** & **status** arguments

```
/* C and C++ */
int MPI_Recv(void *buf, int count,
  MPI_Datatype datatype, int source, int tag,
  MPI_Comm comm, MPI_Status *status);
! Fortran
SUBROUTINE MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG,
  COMM, STATUS, IERROR)
<type> :: BUF(*)
INTEGER :: COUNT, DATATYPE, SOURCE, TAG, COMM, IERROR
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: STATUS
```

## Message Passing - C Example

```
int rank;
MPI_Status status;
float a[10], b[10];
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0){
  MPI_Send(a, 10, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
}
else if (rank == 1){
  MPI_Recv(b, 10, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
  &status);
}
```

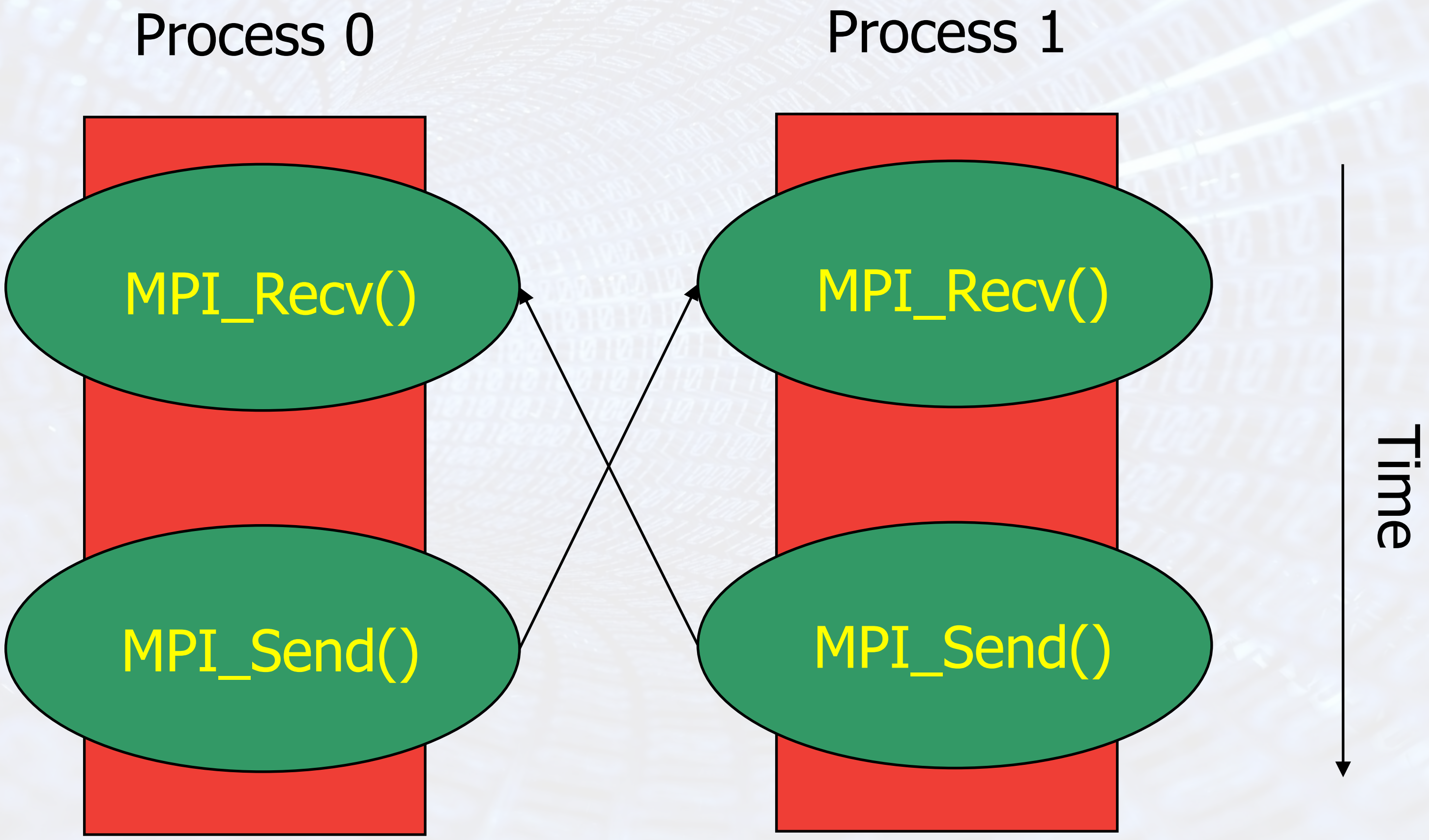
## Message Passing - Fortran Example

```
INTEGER :: rank, ierr
INTEGER, DIMENSION(MPI_STATUS_SIZE) :: status
REAL, DIMENSION(10) :: a, b

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

IF (rank .EQ. 0) THEN
  CALL MPI_SEND(a(1), 10, MPI_REAL, 1, 0, &
  MPI_COMM_WORLD, ierr)
ELSE IF (rank .EQ. 1) THEN
  CALL MPI_RECV(b(1), 10, MPI_REAL, 0, 0, &
  MPI_COMM_WORLD, status, ierr)
END IF
```

## Blocking vs. Non-blocking Communication

- Blocking
  - A blocking communication call will block the execution of the program until the communication is completed
  - **MPI_Send** does not return until the data in the send buffer (i.e. the variable in the user program) can be safely changed.
    - This does not necessarily mean that it's arrived at its destination.
  - **MPI_Recv** does not return until the data in the receive buffer (i.e. the variable in the user program) can be safely accessed.
- Non-blocking
  - A non-blocking communication call will return immediately.
  - It is the user's responsibility to check the completion at a later time.
  - This is useful to: avoid deadlock; overlap communication and computation

## Deadlock



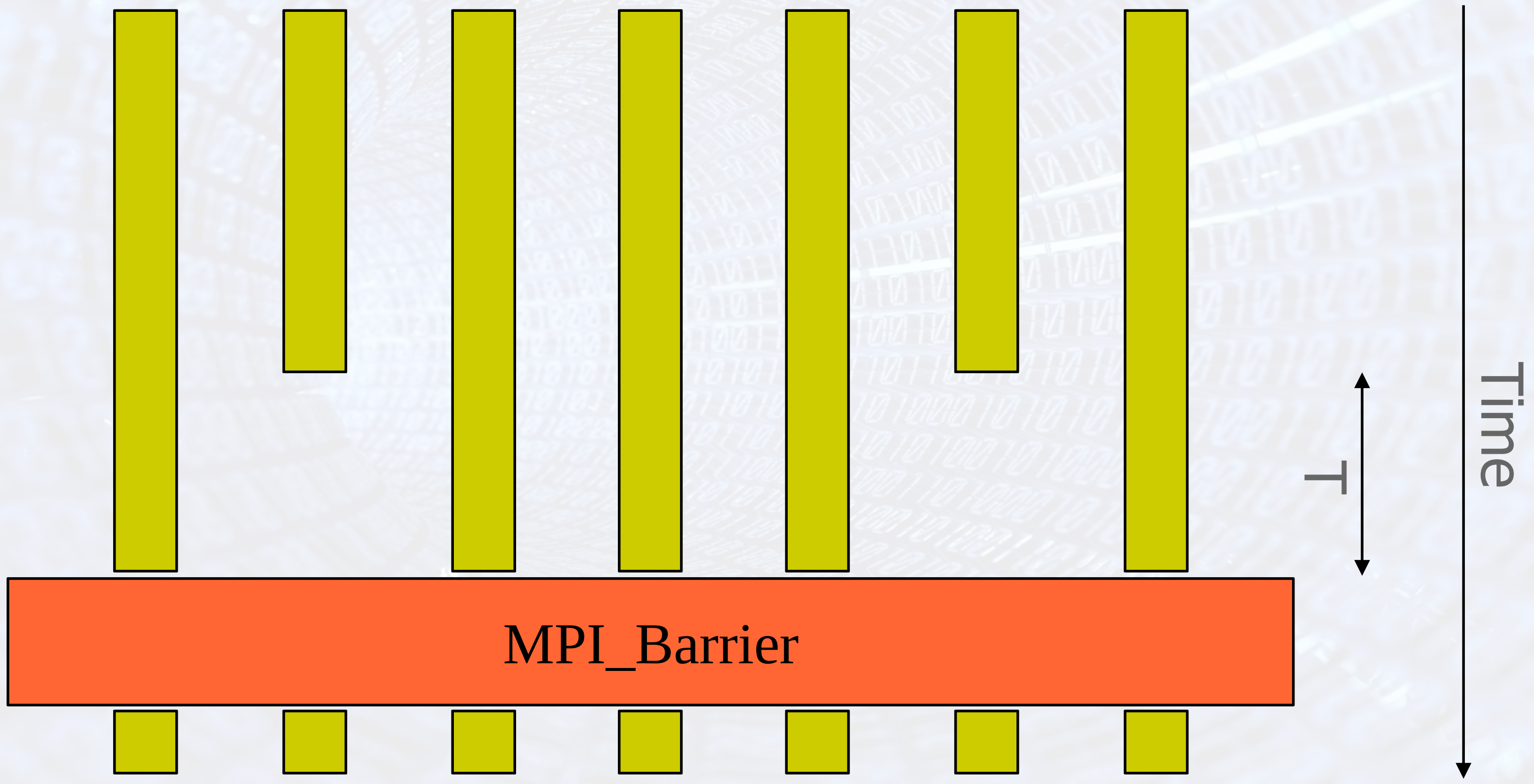
## Non-blocking Communication Example

```
…
if (rank == 0){
  MPI_Isend(a, 10000000, MPI_FLOAT, 1, tag, comm, &request);
/* Do some computation unrelated to a */
  MPI_Wait(&request, &status);
}
else if (rank == 1){
  MPI_Irecv(a, 10000000, MPI_FLOAT, 0, tag, comm, &request);
/* Do some computation unrelated to a */
  MPI_Wait(&request, &status);
}
…
```

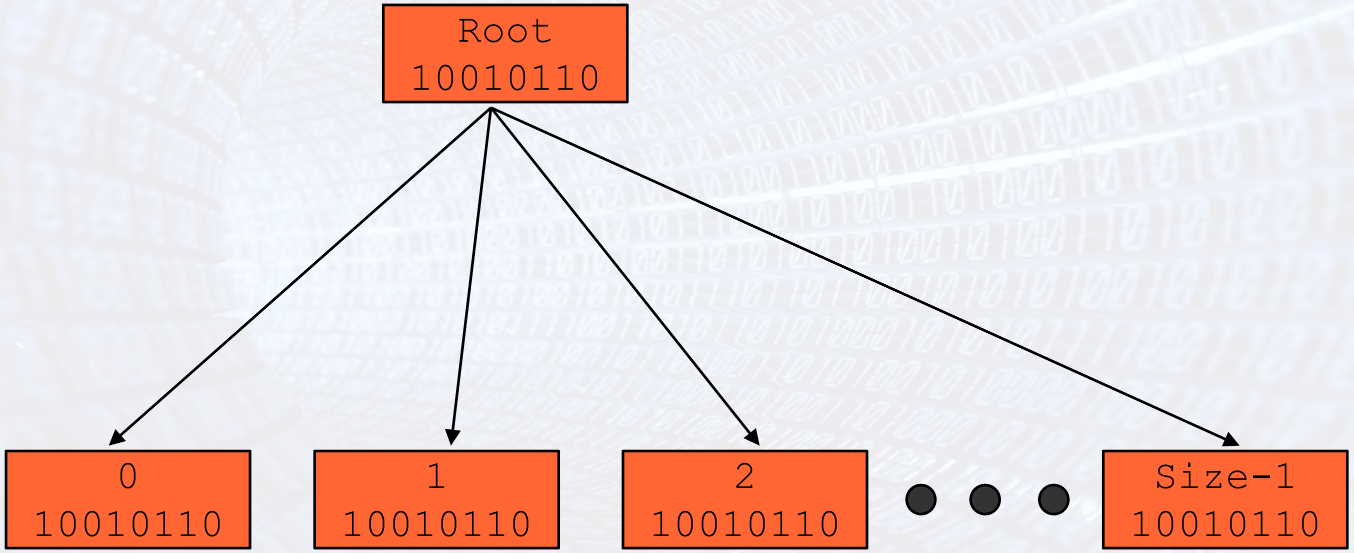
## Point-to-point vs. Collective Communication

- Point-to-point
  - **MPI_Send** and **MPI_Recv** are point-to-point communication routines.
  - There are exactly two processes involved.
- Collective
  - Communication involving a group of (2+) processes is called collective.
  - In theory, you can implement most collective calls using the basic point-to-point communication routines.
  - All collective calls must be made by every process in the group associated with the communicator.
  - Some useful collective operations:
    - Barrier
    - Broadcast, gather/scatter, all-to-all
    - Data reduction

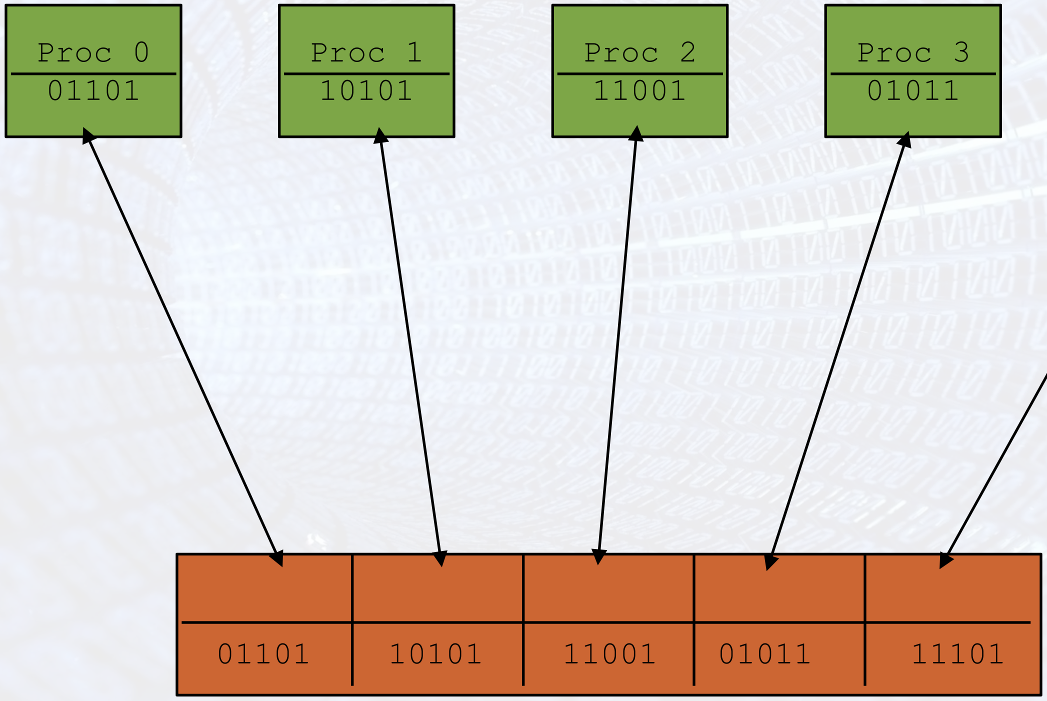
## MPI_Barrier



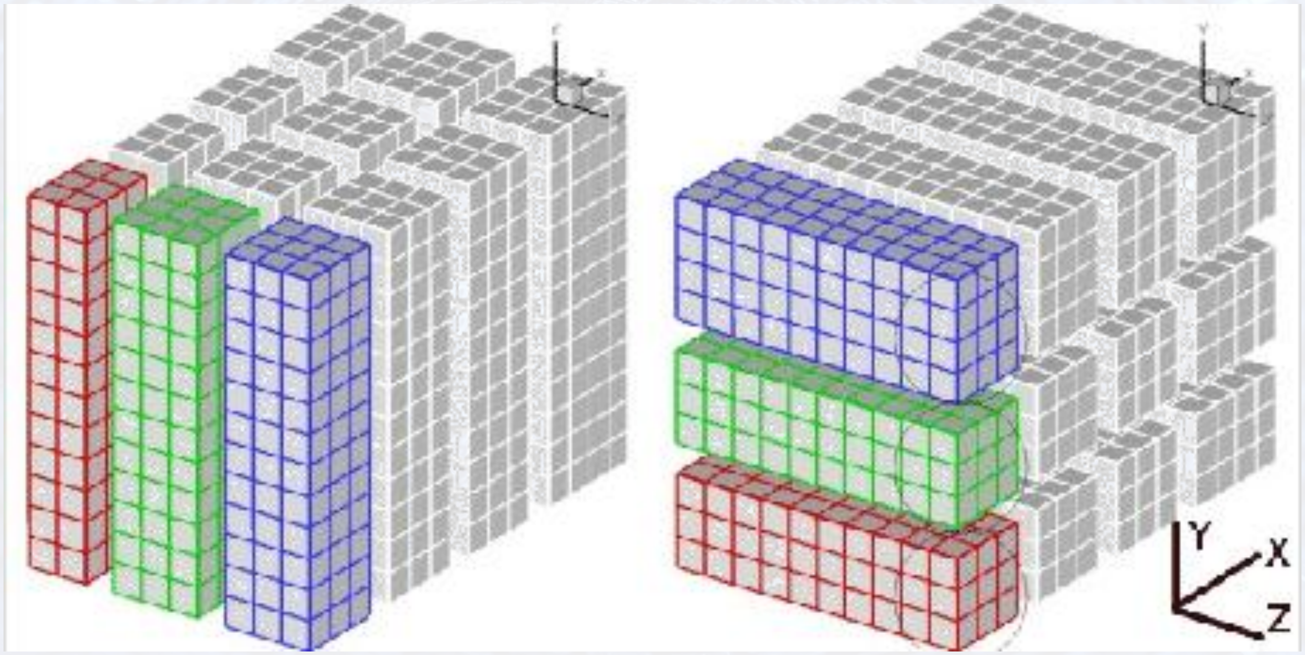
## Broadcast


Root
10010110
0
10010110
1
10010110
2
10010110
Size-1
10010110

## Data Gathering / Scattering



# All-to-all



## Reduction

- Suppose that each process i  has computed a number $X_i$ and that the result needed is the sum of these.

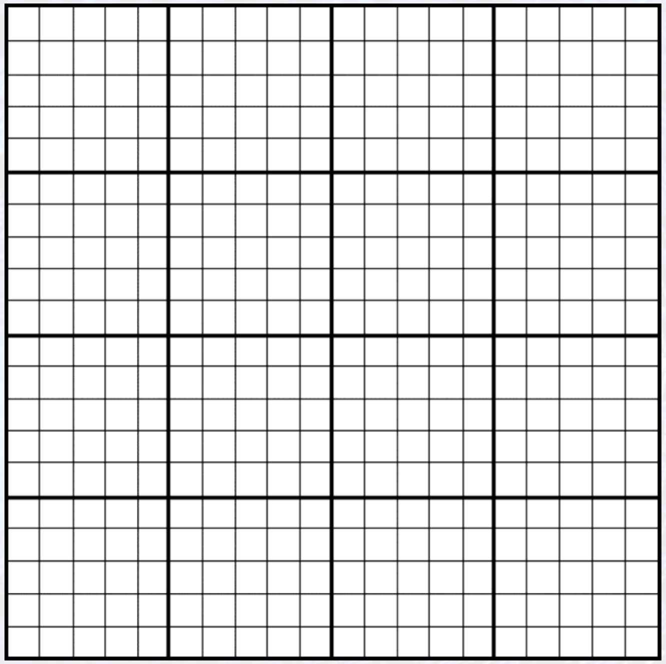
$$X = \sum_{i=0}^{size-1} X_i$$

- This global sum is an example of a reduction operation.
- It combines communication and computation.
- MPI generalises such operation by
  - allowing reductions to proceed element by element on arrays.
  - replacing the sum by an arbitrary associative binary operation.

## Advanced MPI Topics

- This short introduction can only cover the very basics of MPI programming.
- Some of the most useful advanced topics:
  - Groups and communicators
  - Derived datatypes
  - Cartesian topology
  - MPI-IO
- A practical problem is used to demonstrate all these advanced features.

## A Common Matrix-like Dataset



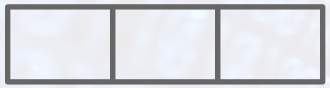
- Groups and communicators – to treat rows/columns specially
- Derived datatypes – to communicate across process boundary more efficiently
- Cartesian topology – to work with neighbouring processes more easily
- MPI-IO – to export and post-process distributed data easily

## Derived Datatypes

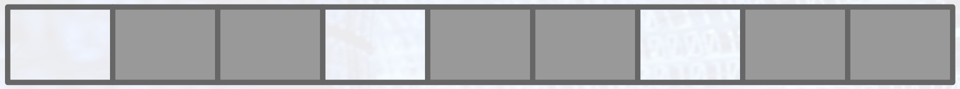
- We have already seen some pre-defined MPI datatypes.
- User-defined derived datatypes can be useful for:
  - Structures in C
  - Types and variables of non-standard size
  - Arrays (in particular those with strided memory pattern)

# Derived Datatypes

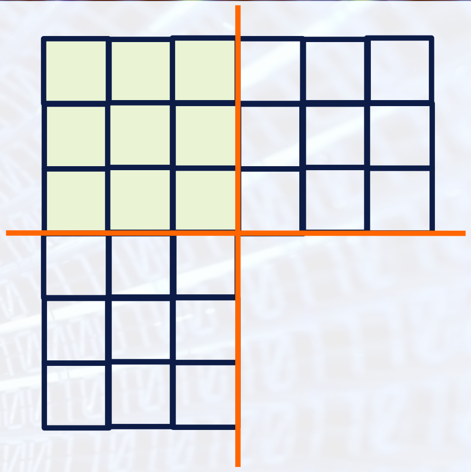
Basic datatype



row_type: a derived datatype that is contiguous



col_type: a derived datatype that is strided

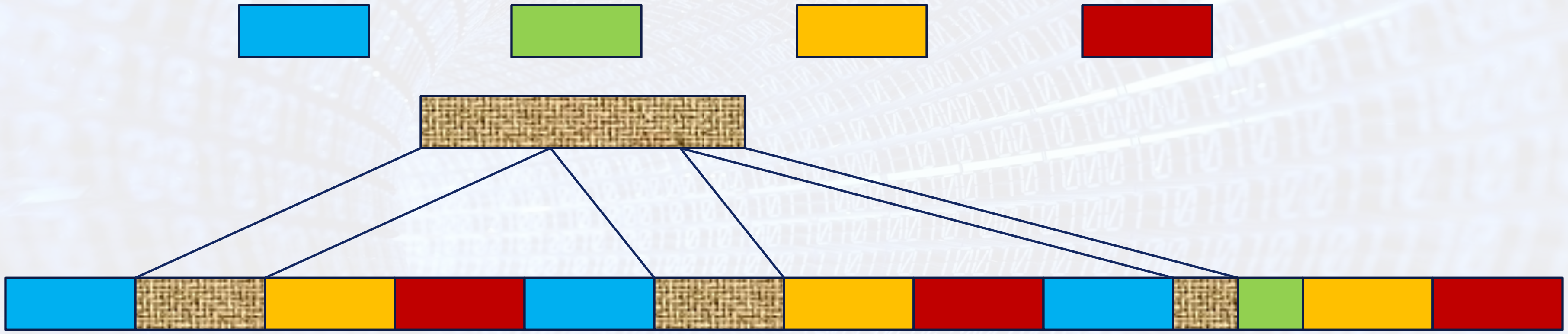


- Use derived datatypes to easily describe matrix rows and columns.
- Derived datatypes, once defined, can be used in communication routines.

```
CALL MPI_SEND(BUF, 3, MPI_FLOAT, dest ……)
CALL MPI_SEND(BUF, 1, row_type, dest ……)
CALL MPI_SEND(BUF, 1, col_type, dest ……)
```

## MPI-IO

- MPI-IO allows a single file to be read or written in parallel by any number of processes.



- It is often the case that a process needs to access several different portions of a file.
- MPI-IO provides routines to facilitate this.
  - e.g. file access patterns described as MPI derived datatypes

## MPI-IO

- There are 100+ MPI-IO routines.
- They are designed very elegantly.
- There are analogies:
  - e.g. blocking/non-blocking communication vs. blocking/non-blocking IO

```
! communication
SUBROUTINE MPI_RECV ( BUF, COUNT, DATATYPE, SOURCE, TAG,
  COMM, STATUS, IERROR )
! IO
SUBROUTINE  MPI_FILE_WRITE ( FH, BUF, COUNT, DATATYPE,
  STATUS, IERROR)
```

## Cartesian Topologies

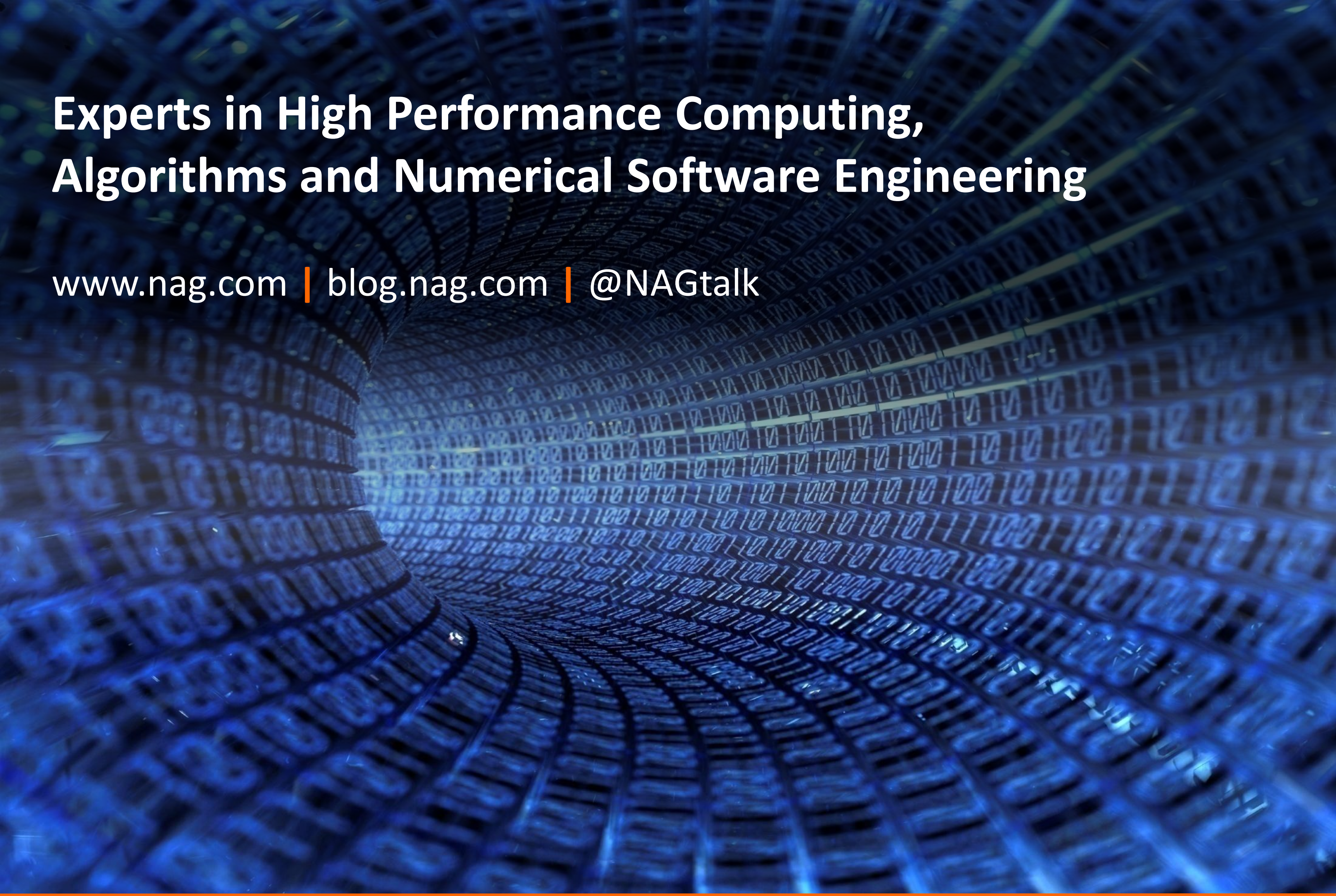
- In a distributed matrix setting, processes are sitting on a Cartesian grid.
  - Therefore they can be referenced easily than using their global ranks.

| | | |
|---|---|---|
| Rank=0 Coord=(0,0) | Rank=1 Coord=(0,1) | Rank=2 Coord=(0,2) |
| Rank=3 Coord=(1,0) | Rank=4 Coord=(1,1) | Rank=5 Coord=(1,2) |

  - Shift operation allows neighbouring processes along process-grid lines to be easily identified.
  - Additional support for periodic boundary condition.

## Further Reading

- The MPI standard documents are available at http://www.mpi-forum.org
  - The specification contains "advice to users" contents.
  - It can be bought as a hardback book.
- Gropp, Lusk and Skjellum, "*Using MPI: Portable Parallel Programming with the Message-Passing Interface*", second edition, The MIT Press.
- Gropp, Lusk and Thakur, "*Using MPI-2: Advanced Features of the Message-Passing Interface*", The MIT Press.



## Experts in High Performance Computing, Algorithms and Numerical Software Engineering

www.nag.com | blog.nag.com | @NAGtalk