Message Passing Interface (MPI)

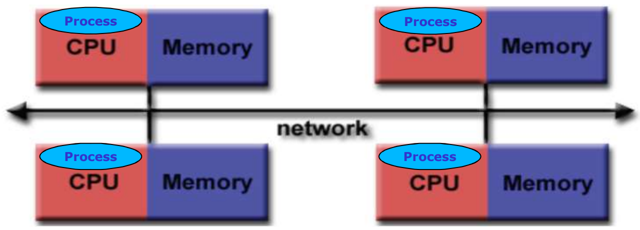
Parallel Programming Approaches

- Approach1: Parallelising compilers and high-level parallel programming languages

- Approach2: Augmenting an existing sequential language with low level constructs expressed by functional calls or compiler directives
  - Parallel programming in C, C++ and Fortran with MPI and OpenMP

Message-Passing Model

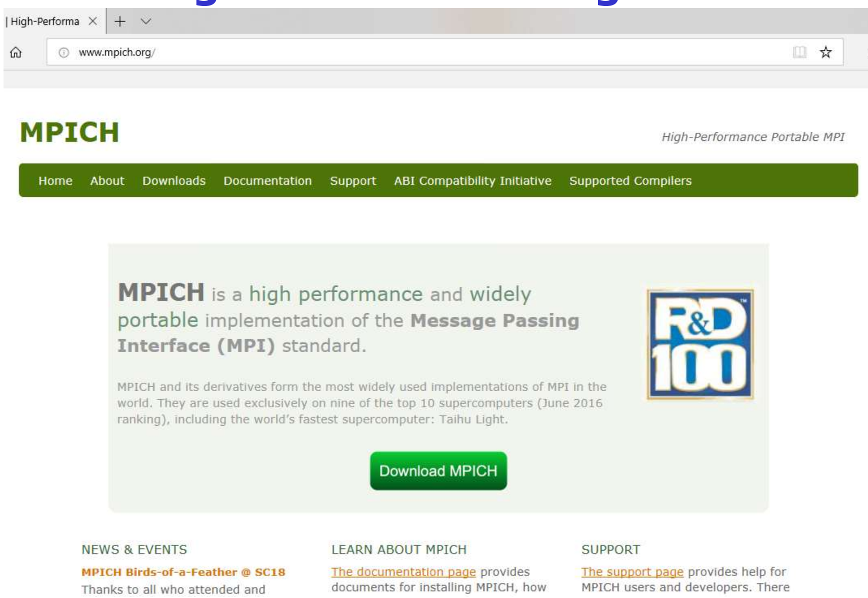
- Collection of processors, each with its own memory
  - Processor has direct access only to the instruction and data stored in local memory
- Interconnection network supports message passing between the processors
- User specifies the number of concurrent processes when the program begins
- Every process executes the same program
- Each process has unique ID number



MPI Standards and Goals

- MPI-1 standard defined in 1994
- Current standard is MPI-3
- Standard gives the uniformity to the code written so that code can be portable, compile and run on any platform that support MPI standard
- Standard specifies the names, calling sequence and subroutines and functions called from C, C++ and Fortran
- Goals:
  - To Provide source code portability
  - To allow efficient implementations across a range of architectures
  - To support heterogeneous parallel architecture

Installing MPICH3 in a Single Machine



Installing MPICH3 in a Single Machine

```
>> tar -xzf mpich-3.3.tar.gz

>> cd mpich-3.3

>> ./configure -disable-fortran

>> sudo make install

>> mpiexec --version
```

https://www.mpi-forum.org/docs/

MPI Programming

- The program begins with pre-processor directive to include header file for MPI

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]){

}
```

  - `argc` and `argv` are needed to pass to the function that initialises MPI
- Some of the MPI function calls:
  - `MPI_Init`: To initialize MPI
  - `MPI_Comm_rank`: To determine process's ID number
  - `MPI_Comm_size`: To find the number of processes
  - `MPI_Finalize`: To shut down the MPI

MPI Overview

- MPI initialization and finalization
- MPI communication environment, size and rank
- MPI datatypes
- Point-to-point communication
  - MPI send and receive
  - Blocking and deadlock
- Collective communication
  - Barrier
  - Broadcast
  - Reduction
  - Gather
  - Scatter
- Time functions

Function `MPI_Init`

- The first MPI function call made by every MPI process
- It allows the system to do any startup needed to handle further calls to the MPI library

```
MPI_Init (&argc, &argv);
```

Function `MPI_Finalize`

- The last MPI function call made by every MPI process i.e. called after a process has completed all its MPI library calls
- It allows the system to free up resources such as memory that has been allocated to MPI

```
MPI_Finalize ();
```


MPI Program Compile and Execution

- Compiling:
  - `mpicc -o example.out example.c`

- Execution:
  - `mpiexec -n <num of processes> ./example.out`

MPI Datatypes

- MPI constants associated with C datatypes:

Name	C datatype
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	Float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Special MPI Datatypes for C

- Special MPI datatypes include:

Name	Meaning
MPI_Comm	Communicator
MPI_Status	Structure containing several pieces of status information for MPI calls

- Communicator: An opaque object that provide the environment for message passing among processes
- When MPI is initialised, every active process becomes a member of a communicator
  - MPI_COMM_WORLD: an MPI constant – a default communicator

Function `MPI_Comm_rank`

- Processes within a communicator are ordered
- Rank: Position of a process in the overall order
- Each process has a unique rank (ID number)
- Function to determine rank of processes within a communicator

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

Function `MPI_Comm_size`

- To determine total number of processes in a communicator

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


MPI Overview

- MPI initialization and finalization
- MPI communication environment, size and rank
- MPI datatypes
- Point-to-point communication
  - MPI send and receive
  - Blocking and deadlock
- Collective communication
  - Barrier
  - Broadcast
  - Reduction
  - Gather
  - Scatter
- Time functions

Point-to-Point Communication

- One process send message and other process receive that message

- For point-to-point communication we need to specify source and destination
- Any message that is passed will have envelope and message body

Point-to-Point Communication

- Envelope: Contain source and destination address
  - Source : The sending process
  - Destination : The receiving process
  - Communicator: Specifies a group of processes to which both sender and receiver belongs
  - tag : used to classify the messages
- Message body:
  - Buffer : The message data (Buffer as an array)
  - Datatype : The type of message data
  - Count : The number of items of type datatype in buffer

Function `MPI_Send`

```
int MPI_Send (void *buff, int count,
MPI_Datatype dtype, int dest, int tag, MPI_Comm
comm);
```

•Blue coloured arguments are message body

•Red coloured arguments are envelope

•All arguments are input arguments

•An error code is returned by the function

Function `MPI_Recv`

```
int MPI_Recv (void *buff, int count,
MPI_Datatype dtype, int source, int tag,
MPI_Comm comm, MPI_Status *status);
```

•`buff` and `status` are output arguments

•Rest are all input arguments

Completion and Blocking in Point-to-Point Communication

- Completion for a call `MPI_Recv`:
  - Matching message has arrived and the message data have been copied into the output argument of the call
- Completion for a call `MPI_Send`:
  - Message specified in the input argument of the call has been handed off to MPI
    - Variable passed as input argument can now be over written and reused
    - The message is copied to internal buffer for later delivery
- MPI will wait for the destination process to receive the message
- Blocking:
  - When the message passed through `MPI_Send` is larger than the available buffer at receiving side, sending process will be blocked till the receiving process start receiving or sufficient buffer is available

Deadlock and Resolving Deadlock

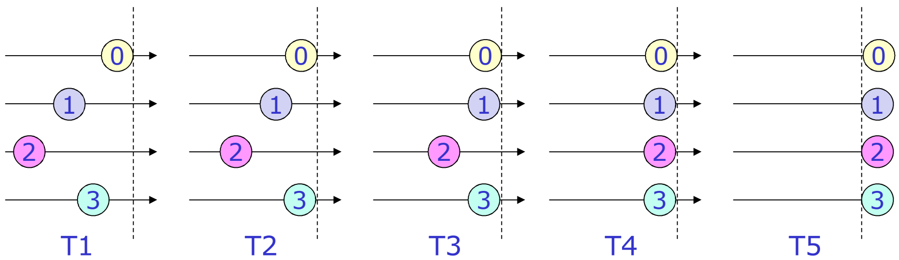
- Deadlock:
  - Occurs when two or more processes are blocked
  - Each is waiting for the other to make progress
- Example:
  - Process 0 cannot proceed until process 1 sends a message
  - Process 1 cannot proceed until process 0 sends a message
- Resolving Deadlock:
  - Process 0 receives from process 1 and then sends the message to process 1
  - Process 1 sends message to process 0 and then received from process 0
- Sending the message depends on the available buffer in the receiving side

MPI Overview

- MPI initialization and finalization
- MPI communication environment, size and rank
- MPI datatypes
- Point-to-point communication
  - MPI send and receive
  - Blocking and deadlock
- Collective communication
  - Barrier
  - Broadcast
  - Reduction
  - Gather
  - Scatter
- Time functions

Collective Communication

- Communication operation in which a group of processes work together to distribute or gather together a set of one or more values
- One-to-many or many-to-one communication
- **Barrier Synchronization**:
  - Synchronising all the processes in the communicator

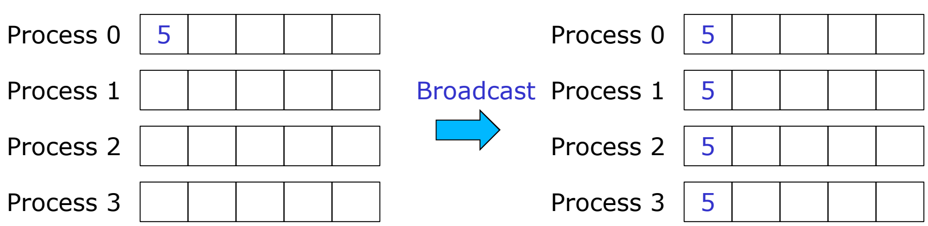


- **Function `MPI_Barrier`**:

```
int MPI_Barrier (MPI_Comm comm);
```


Broadcast

- Enables a process to broadcast one or more data items of the same type to all other processes in the communicator

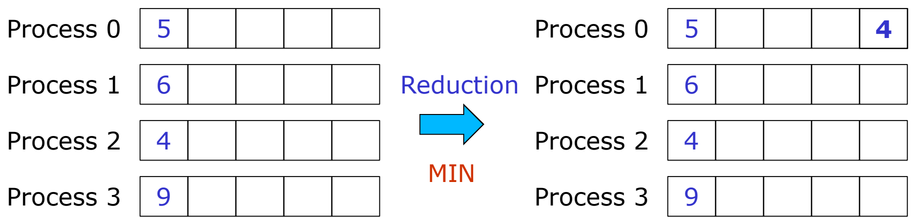


- **Function `MPI_Bcast`**:

```
int MPI_Bcast (
   void           *buffer,  // Address of first data item to be broadcasted
   int            count,    // number of elements to be broadcasted
   MPI_Datatype   dType,    // Type of element to be broadcasted
   int            root,     // ID (rank) of the process doing broadcast
   MPI_Comm       comm);    // Communicator
```

Reduction

- Performs one or more reduction operations on the values submitted from all the processes in a communicator



- **Function `MPI_Reduce`:**

```
int MPI_Reduce (
   void         *send_buffer,  // Address of first reduction element
   void         *recv_buffer,  // Address of reduction result
   int           count,     // number of elements to be send or reduction
   MPI_Datatype dType,     // Type of element
   MPI_Op        Operator  // Reduction operator
   int           root,     // ID (rank) of the process receiving
   MPI_Comm      comm);    // Communicator
```


Reduction Operators

- MPI's built-in reduction operators:

Name	Meaning
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise exclusive or
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or
`MPI_MAX`	Maximum
`MPI_MAXLOC`	Maximum and its location
`MPI_MIN`	Minimum
`MPI_MINLOC`	Minimum and its location
`MPI_PROD`	Product
`MPI_SUM`	Sum

Gather

- Enables each process in a communicator to send the contents of its send buffer to the root process
- Root process receives the messages and store them in rank order

						Gather						
Process 0	5						Process 0	**5**	**6**	**4**	**9**	
Process 1	6					→	Process 1	6				
Process 2	4						Process 2	4				
Process 3	9						Process 3	9				

- **Function `MPI_Gather`:**

```
int MPI_Gather (
   void          *send_buffer,  // Starting address of send buffer (IN)
   int            send_count,  // number of elements in send buffer (IN)
   MPI_Datatype send_dType,  // Type of send buffer element (IN)
   void          *recv_buffer,  // Starting address of receiving buffer (OUT)
   int            recv_count,  // number of elements in recv buffer (IN)
   MPI_Datatype recv_dType,     // Type of receiving buffer element (IN)
   int            root,     // ID (rank) of the process doing gathering (IN)
   MPI_Comm       comm);    // Communicator (IN)
```


Scatter

- A group of elements held by the root process is divided into equal chunks and one chunk is send to every process in the communicator including root.

						Scatter						
Process 0	5	6	4	9			Process 0	5				
Process 1						→	Process 1	6				
Process 2							Process 2	4				
Process 3							Process 3	9				

- **Function `MPI_Scatter`:**

```
int MPI_Scatter (
   void          *send_buffer,  // Starting address of send buffer (IN)
   int       send_count,  // number of elements in send by each process (IN)
   MPI_Datatype send_dType,  // Type of send buffer element (IN)
   void          *recv_buffer,  // Starting address of receiving buffer (OUT)
   int            recv_count,  // number of elements in recv buffer (IN)
   MPI_Datatype recv_dType,   // Type of receiving buffer element (IN)
   int            root,     // ID (rank) of the process doing gathering (IN)
   MPI_Comm       comm);      // Communicator (IN)
```

MPI Overview

- MPI initialization and finalization
- MPI communication environment, size and rank
- MPI datatypes
- Point-to-point communication
  - MPI send and receive
  - Blocking and deadlock
- Collective communication
  - Barrier
  - Broadcast
  - Reduction
  - Gather
  - Scatter
- **Time functions**

Time Functions for Measuring Performance

- One way to measure the performance of a parallel application is to look at the wall clock time.
- It measures the number of seconds that elapse from the time we initiate the execution until the program terminates
- **Function `MPI_Wtime`:**
  - Returns the number of seconds that have elapsed since some point of time in the past

```
double MPI_Wtime (void);
```

- **Function `MPI_Wtick`:**
  - Returns the precision of the result returned by `MPI_Wtime`

```
double MPI_Wtick (void);
```

Example: Inneproduct Between Two Vector

- Let $\mathbf{x}=[x_1,\ x_2,\ldots,\ x_d]^\top$ and $\mathbf{y}=[y_1,\ y_2,\ldots,\ y_d]^\top$ be two vectors
- Innerproduct between two vector is given as:

$$z = \mathbf{x}^\top \mathbf{y}$$

$$z = \sum_{i=1}^{d} (x_i\ y_i)$$

- Assignment:
  - Read or Create the vectors in root processor (i.e. id=0)
  - Use scatter function
  - Use reduce function

Text Books

- https://www.mpi-forum.org/docs/
- M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill, 2004.