Introduction to MPI & Parallel Programming

- Parallelisms and Decomposition.
  - Data Decomposition.
  - Domain Decomposition.
  - Master–Slave model.
  - Pipelining.
  - Functional Decomposition.

Introduction to MPI & Parallel Programming

- MPI as a tool for parallel programming.
  - Initializing MPI.
  - Point to Point communications.
  - Global Communications.
  - Communicators and Topology.
  - Derived data types and Data packing.
  - Parallel I/O (?)

Parallelisms and Decomposition: Data Decomposition

- Large data sets can be divided into smaller parts and analysis of these can be carried out on compute nodes.

  - If the analysis of subsets of data can be carried out independent of the rest of data, the problem can be classified as an embarrassingly parallel problem.

  - If the data size is not very large but extensive calculations are required to process it, all the data can be made available at all the nodes.

Parallelisms and Decomposition: Domain Decomposition

- Domain decomposition refers to the special case where data is description of quantities in an n-dimensional space, e.g.,
  - Partial differential equations on a grid.
  - Cellular automata.

Parallelisms and Decomposition: Master-Slave Model

- A number of *almost* independent tasks are distributed to compute nodes. One node, designated as the master node distributes tasks to other (slave) nodes and collects the result. Typically the number of tasks is much larger than the number of slave nodes.
  - Good for *embarrassingly parallel* problems, e.g., Monte-Carlo methods, analysis of a number of similar data sets or models, etc.
  - Not very efficient for generic problems.
  - Easier to make the system fault tolerant, e.g., PVM.

Parallelisms and Decomposition: Pipelining

- A series of operations to be carried out on the given data can be parallelized by assigning these operations to different compute nodes. Data is passed from one node to the next till all the operations have been carried out.
  - Not very efficient unless the network is very fast.
  - There is significant overhead if the data set is not large.
  - This is used in most modern day processors to increase the effective computing power.

Parallelisms and Decomposition: Functional Decomposition

- If the required computation involves two sets of operations to be carried out, and these can be carried out concurrently, then these operations can be assigned to different sets of compute nodes. The final answer is obtained by combining the result of the two calculations.
  - This type of parallelism is not encountered very often as most algorithms/computations involve a sequence of operations.

Passing Messages for Parallel Programming

- MPI is based on the premise that all the models of parallel decomposition of problems can be implemented by passing messages between different processes.
    - Processes do not share memory/address space.
    - Each process is identified by a unique label or *id*.
    - There is a well defined set of processes.
    - Data can be shared by sending it as a message.
    - Processes can synchronize computation by passing messages.
    - Message passing is programmed and controlled by the user.

MPI: The role of process id

- Processes have numerical id ranging from 0 to $n-1$, where n is the total number of processes that are invoked.
- The process id is used as the address for sending messages.
- The process id is used to make different compute nodes carry out different tasks using the same code.
- Subsets of processes can be defined.

Datatypes in MPI

- In order to facilitate transfer of data between compute nodes with different operating systems, `MPI` uses its own set of data types.
  - These correspond to native data types.
  - This allows conversion between systems with little-endian and big-endian systems.

- MPI also defines a set of variables for its own operations. These and the MPI data types are defined together in `mpi.h`, `mpif.h` and `mpi.mod` for C, FORTRAN 77 and FORTRAN 90/95, respectively.

Initializing MPI

- Program should be compiled with the module/include files and the appropriate MPI libraries must be linked.
- The user supplies the list of compute nodes where the MPI program is to run. Password free access to these nodes is required.
- The number of processes to be launched are also specified at run time. This number can be smaller than, equal to, or larger than the number of compute nodes specified.

Sample Program

```
program test1

implicit none
include 'mpif.h'
integer id,nprocs,ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,id,ierr)
write(*,*)'Vanakkam from process ',id,'of',nprocs,'processes.'
call MPI_FINALIZE(ierr)

end
```

MPI: Introduction

- Standard output, input and error of all processes is connected to the corresponding channels for the process with rank 0.
  - Should not use standard input in MPI programs.
  - Should use error trapping for production runs.
  - Standard output is also a bad idea in general.
  - You must use `MPI_FINALIZE` for graceful exit.

MPI: Using `id` to divide work

```
if (id .eq.  0) then
      write(*,*)’namaste from process’,id
else if (id .eq.  1) then
      write(*,*)’Vanakkam from process’,id
else
      write(*,*)’Hello World from process’,id
end if
```

MPI: Sending and Receiving Messages

- Many commands are available for point to point communications. The basic commands are:

```
MPI_SEND(buffer,count,datatype,destination,tag,communicator)
```

```
MPI_RECV(buffer,count,datatype,source,tag,communicator,status)
```

  - Tag can be used to differentiate between communications between the same pair of processes.
  - Status gives information like the number of elements received.
  - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` can be used to accept communications from any source, or with any tag. Status give information about the actual source and tag for the message.

MPI: Deadlocks

- Send and Recv are blocking communications, i.e., control does not pass to the next statement unless the message being sent has been received.

  - Unless one is careful, this can easily lead to deadlocks.

```
if (id .eq.  0) then
      call MPI_SEND(buffer,count,datatype,1,.  .  .  .)
      call MPI_RECV(buffer,count,datatype,1,.  .  .  .)
else if (id .eq.  1) then
      call MPI_SEND(buffer,count,datatype,0,.  .  .  .)
      call MPI_RECV(buffer,count,datatype,0,.  .  .  .)
end if
```

MPI: Pitfalls

- Consider a situation in which a chain of processes are expected to exchange messages.

  - Each process send a message to the next higher process (in terms of id), except the last process.
  - Send for the process with $id = n - 2$ will finish first.
  - It can then receive message from $id = n - 3$, and so on.
  - If n is large, this process will take a long time to finish.

MPI: Problems for tutorial session

- Compare methods for exchanging messages with neighbors, for the situation described above.

- Send and Receive are supposed to be blocking communications. Test whether this is independent of the message size.

MPI: Point to Point communications

- Asynchronous equivalents of send and receive are available. These are useful for avoiding deadlocks and wait time.

  - From personal experience, the combination of `MPI_ISEND` and `MPI_RECV` is the best. Here the sender sends the message and gets on with other work. Receive can be located at an appropriate point so that waiting is not required.

  - These require a buffer to store communicated data.

- `MPI_ANY_SOURCE` is very useful for implementing the master-slave model.

MPI: Series Summation

```
Each process computes a partial sum.

Partial sum is sent to the parent node (0).

Parent node adds the partial sums to get the result.
```

- Compute the value of π by summing a series representation.

MPI: Performance Evaluation

• If T_n is the time taken by a program on n processors and T_{seq} is the time taken by the equivalent sequential program, we can define speedup as

$$S_n \equiv \frac{T_{seq}}{T_n} \tag{1}$$

• Efficiency of parallelization is defined as:

$$E_n \equiv S_n \times n \tag{2}$$

• If a fraction ϵ of the program cannot be parallelized then the maximum speedup attainable is $1/\epsilon$ (*Amdahl's Law*).

$$S_n = \frac{T_{seq}}{T_n} = \frac{1}{\epsilon + (1-\epsilon)/n} \tag{3}$$

MPI: Global Communications

- Communications involve all the processes in a given communicator.
  - MPI calls for global communications must be visible on all processes, unlike calls for point to point communications.
  - One to many, many to one, and all to all communications are available.
  - In the process of these communications, data can be sent as a whole, split, or even combined.

MPI: Broadcast & Reduce

- MPI_BCAST(buffer, count, data type, root, communicator)
- MPI_REDUCE(send buffer, recv buffer, count, data type, operation, root, communicator)
    - MPI_SUM
    - MPI_PROD
    - MPI_MIN
    - MPI_MAX

Problems

- Rewrite the series summation program using `MPI_REDUCE`.
- Write a program to do Monte-Carlo integration of the curve $y = \sqrt{1 - x^2}$, $0 \leq x \leq 1$ in parallel. Supply the value of random seed to processes using `MPI_BCAST` and recover the final solution using `MPI_REDUCE`. Use this to find the value of π. Take care to avoid using the same random numbers on multiple processes.

MPI: Broadcast & Reduce

- For n processors, these take `ceiling(log_2(n))` times the time taken for a single `MPI_SEND` with the same amount of data.
  - Thus global communications should be used as much as possible when working with a large number of processors.