Message Passing Programming

- *Message Passing* is the most widely used parallel programming model.

- Message passing works by creating a number of processes, uniquely named, that interact by sending and receiving messages to and from one another (hence the *message passing*).
  - Generally, processes communicate through sending the data from the address space of one process to that of another.
    - Communication of processes (via files, pipe, socket)
    - Communication of threads within a process (via global data area)

- Programs based on message passing can be based on standard sequential language programs (C/C++, Fortran), augmented with calls to library functions for sending and receiving messages.
Message Passing Interface (MPI)

- There are different message passing models; MPI is one of the most popular models (PVM is another one);

- MPI is a specification, not a particular implementation
  - Does not specify process startup, error codes, amount of system buffer, etc

- MPI implementation is a library, not a language
  - There are different MPI implementations: MPICH, LAM/MPI, OPEN MPI

- Message passing model > MPI specification > MPI implementation
OpenMP vs MPI

In a nutshell

MPI is used on distributed-memory systems

OpenMP is used for code parallelisation on shared-memory systems

- Both are explicit parallelism
- High-level control (OpenMP), lower-level control (MPI)
A little history

- Message-passing libraries developed for a number of early distributed memory computers
- By 1993 there were loads of vendor specific implementations
- By 1994 MPI-1 came into being
- By 1996 MPI-2 was finalized
The MPI programming model

- **MPI standards** -
  - MPI-1 (1.2), MPI-2 (2.0)

- **Standard bindings** - for C, C++ and Fortran. Have seen MPI bindings for Python, Java etc (non-standard)

- We will stick to the C binding, for the lectures and coursework. More info on MPI [www.mpi-forum.org](http://www.mpi-forum.org)

- **Implementations** - For your laptop pick up MPICH (free portable implementation of MPI) [http://www-unix.mcs.anl.gov/mpi/mpich/index.htm](http://www-unix.mcs.anl.gov/mpi/mpich/index.htm)
MPI

MPI is a complex system comprising of 129 functions with numerous parameters and variants.

Six of them are indispensable, but can write a large number of useful programs already.

Other functions add flexibility (datatype), robustness (non-blocking send/receive), efficiency (ready-mode communication), modularity (communicators, groups) or convenience (collective operations, topology).

In the lectures, we are going to cover most commonly encountered functions.
The MPI programming model

- Computation comprises one or more processes that communicate via library routines and sending and receiving messages to/from other processes

- (Generally) a fixed set of processes created at outset, one process per processor
  - Different from PVM
Intuitive Interfaces for sending and receiving messages

- Send(data, destination), Receive(data, source)
  
  - minimal interface

- Not enough in some situations, we also need
  
  - Message matching – add message_id at both send and receive interfaces

  - they become Send(data, destination, msg_id), receive(data, source, msg_id)

- Message_id
  
  - Is expressed using an integer, termed as message tag
  
  - Can differentiate the messages from the same process
  
  - Enable the messages to be processed in an ordered fashion
How to express the data in the send/receive interfaces

→ Early stages:
  - (address, length) for the send interface
  - (address, max_length) for the receive interface

→ They are not always good
  - The data to be sent may not be in the contiguous memory locations
  - Storing format for data may not be the same or known in advance in heterogeneous platform

→ Eventually, a triple (address, count, datatype) is used to express the data to be sent and (address, max_count, datatype) for the data to be received
  - Reflecting the fact that a message contains much more structures than just a string of bits, For example, (vector_A, 300, MPI_REAL)
  - Programmers can construct their own datatype

→ Now, the interfaces become send(address, count, datatype, destination, msg_id) and receive(address, max_count, datatype, source, msg_id)
How to distinguish messages

- Message tag is necessary, but not sufficient
- So, communicator is introduced ...
Communicators

- Messages are put into contexts
  - Contexts are allocated at run time by the system in response to programmer requests
  - The system can guarantee that each generated context is unique
- The processes belong to groups
- The notions of context and group are combined in a single object, which is called a communicator
  - A communicator identifies a group of processes and a communication context
  - The MPI library defines a initial communicator, MPI_COMM_WORLD, which contains all the processes running in the system
  - The messages from different process groups can have the same tag
- So the send interface becomes send(address, count, datatype, destination, tag, comm)
Status of the received messages

→ The structure of the message status is added to the receive interface

→ Status holds the information about source, tag and actual message size
  - In the C language, source can be retrieved by accessing status.MPI_SOURCE,
  - tag can be retrieved by status.MPI_TAG and
  - actual message size can be retrieved by calling the function MPI_Get_count(&status, datatype, &count)

→ The receive interface becomes receive(address, maxcount, datatype, source, tag, communicator, status)
How to express source and destination

- The processes in a communicator (group) are identified by ranks.

- If a communicator contains $n$ processes, process ranks are integers from 0 to $n-1$.

- Source and destination processes in the send/receive interface are the ranks.
Some other issues

In the receive interface, tag can be a wildcard, which means any message will be received.

In the receive interface, source can also be a wildcard, which match any source.
MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf address of send buffer
count no. of elements to send (>=0)
datatype of elements
dest process id of destination
tag message tag
comm communicator (handle)
MPI basics

First six functions (C bindings)

**MPI_Send (buf, count, datatype, dest, tag, comm)**

Send a message

- **buf**: address of send buffer
- **count**: no. of elements to send \((\geq 0)\)
- **datatype**: of elements
- **dest**: process id of destination
- **tag**: message tag
- **comm**: communicator (handle)
MPI basics

First six functions (C bindings)

MPI_Send (buf, count, datatype, dest, tag, comm)

Send a message

buf address of send buffer
count no. of elements to send (>=0)
datatype of elements
dest process id of destination
tag message tag
comm communicator (handle)
MPI basics

First six functions (C bindings)

**MPI_Send** *(buf, count, datatype, dest, tag, comm)*

*Calculating the size of the data to be send ...*

- **buf** | address of send buffer
- **count * sizeof (datatype)** | bytes of data
**MPI basics**

First six functions (C bindings)

```c
MPI_Send (buf, count, datatype, dest, tag, comm)
```

*Send a message*

- **buf**: address of send buffer
- **count**: no. of elements to send (>=0)
- **datatype**: of elements
- **dest**: process id of destination
- **tag**: message tag
- **comm**: communicator (handle)
MPI basics

First six functions (C bindings)

**MPI_Send (buf, count, datatype, dest, tag, comm)**

(send a message)

- **buf**: address of send buffer
- **count**: no. of elements to send (>=0)
- **datatype**: of elements
- **dest**: process id of destination
- **tag**: message tag
- **comm**: communicator (handle)
MPI basics

First six functions (C bindings)

MPI_Recv (buf, count, datatype, source, tag, comm, status)

Receive a message

- **buf**: address of receive buffer (var param)
- **count**: max no. of elements in receive buffer (>=0)
- **datatype**: of receive buffer elements
- **source**: process id of source process, or MPI_ANY_SOURCE
- **tag**: message tag, or MPI_ANY_TAG
- **comm**: communicator
- **status**: status object
MPI basics

First six functions (C bindings)

MPIL_Init (int *argc, char ***argv)

Initiate a computation

argc (number of arguments) and argv (argument vector) are main program’s arguments

Must be called first, and once per process

MPI_Finalize ( )

Shut down a computation

The last thing that happens
MPI basics

First six functions (C bindings)

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

*Determine number of processes in comm*

*comm* is communicator handle, *MPI_COMM_WORLD* is the default (including *all* MPI processes)

*size* holds number of processes in group

**MPI_Comm_rank (MPI_Comm comm, int *pid)**

*Determine id of current (or calling) process*

*pid* holds id of current process
MPI basics – a basic example

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    printf("Hello, world. I am %d of %d\n", rank, nprocs);
    MPI_Finalize();
}
```

mpirun -np 4 myprog
Hello, world. I am 1 of 4
Hello, world. I am 3 of 4
Hello, world. I am 0 of 4
Hello, world. I am 2 of 4
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int rank, size, i;
    int buffer[10];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (size < 2)
    {
        printf("Please run with two processes.\n");
        MPI_Finalize();
        return 0;
    }
    if (rank == 0)
    {
        for (i=0; i<10; i++)
        {
            buffer[i] = i;
            MPI_Send(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
        }
    }
}
if (rank == 1)
{
    for (i=0; i<10; i++)
        buffer[i] = -1;
    MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    for (i=0; i<10; i++)
    {
        if (buffer[i] != i)
            printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);
    }
    MPI_Finalize();
}
MPI language bindings

- Standard (accepted) bindings for Fortran, C and C++
- Java bindings are work in progress
  - JavaMPI: Java wrapper to native calls
  - mpiJava: JNI wrappers
  - jmpi: pure Java implementation of MPI library
  - MPIJ: same idea
- Java Grande Forum trying to sort it all out
- We will use the C bindings