High Performance Computing

Course Notes 2009-2010

Message Passing Programming I
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, DSM).
    - 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
  - specify a number of routines as well as the parameters that the routines should have
  - 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, OpenMPI

- 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
- OpenMP is higher-level control
  - Compiler can automatically parallelise the codes when instructed
- MPI is lower-level control
  - Data partition, allocation and communication are conducted by programmers
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
  - Emphasize message passing
  - Has a static runtime environment
- By 1996 MPI-2 was finalized
  - includes new features such as
    - parallel I/O,
    - dynamic process management
    - remote memory operations
The MPI programming model

- **MPI standards** -
  - MPI-1, MPI-2

- **Standard bindings** - for C, C++ and Fortran.

- There are MPI bindings for Python, Java etc (non-standard)

- We will stick to the C binding, for the lectures and coursework.
MPI functions

MPI is a complex system comprising of numerous functions with various 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
Intuitive Interfaces for sending and receiving messages

→ Send(data, destination), Receive(data_loc, 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 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_tag) and receive(address, max_count, datatype, source, msg_tag)
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 in a MPI system are divided into groups

- The notions of context and group are combined in a single object, which is called a communicator
  - A communicator consists of 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)

\[
\text{MPI\_Send}(\text{buf, count, datatype, dest, tag, comm})
\]

*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)

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)

\[ \text{MPI\_Send} (\text{buf, count, datatype, dest, tag, comm}) \]

*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 ($\geq 0$)
- **datatype**: of elements
- **dest**: process id of destination
- **tag**: message tag
- **comm**: communicator (handle)
First six functions (C bindings)

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

Receive a message

- **buf**: address of receive buffer
- **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)

MPI_Init (int *argc, char **argv)

*Initiate a MPI computation*

*argc* (number of arguments) and *argv* (argument vector) hold 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();
}
```

```bash
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);
    }
}
MPI basics – send and recv example (2)

```c
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++**

- **Two types of Java bindings (work in progress)**
  - **native MPI bindings**: native MPI library is called by Java programs through Java wrappers
    - **JavaMPI**
    - **mpiJava**
  - **pure Java implementations**: the whole MPI library is rewritten in Java
    - **jmpi**
    - **MPIJ**

- **Java Grande Forum trying to sort it all out**

- **We will use the C bindings**
Modularity

- MPI supports modular programming via *communicators*
- Provides information hiding by encapsulating **local communications** and having **local namespaces for processes**
- All MPI communication operations specify a communicator (process group that is engaged in the communication)
Forming new communicators – one approach

MPI_Comm world, workers;
MPI_Group world_group, worker_group;
int ranks[1];
MPI_Init(&argc, &argv);
world=MPI_COMM_WORLD;
MPI_Comm_size(world, &numprocs);
MPI_Comm_rank(world, &myid);
server=numprocs-1;

MPI_Comm_group(world, &world_group);
ranks[0]=server;

MPI_Group_excl(world_group, 1, ranks, &worker_group);
MPI_Comm_create(world, worker_group, &workers);
MPI_Group_free(&world_group);
MPI_Comm_free(&workers);
Forming new communicators - functions

int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

Int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)

int MPI_Group_free(MPI_Group *group)

int MPI_Comm_free(MPI_Comm *comm)
Forming new communicators – another approach (1)

**MPI_Comm_split (comm, colour, key, newcomm)**

*Create one or more new communicators from the original comm*

- **comm**: communicator (handle)
- **colour**: control of subset assignment (processes with same colour are in same new communicator)
- **key**: control of rank assignment
- **newcomm**: new communicator

Is a *collective* communication operation (must be executed by all processes in the *comm*)

Is used to (re-) allocate processes to communicator (groups)
Forming new communicators – another approach (2)

MPI_Comm_split (comm, colour, key, newcomm)

MPI_Comm comm, newcomm; int myid, color;
MPI_Comm_rank(comm, &myid); // id of current process
color = myid%3;
MPI_Comm_split(comm, colour, myid, *newcomm);
Forming new communicators – another approach (3)

MPI_Comm_split (comm, colour, key, newcomm)

- New communicator created for each new value of \textit{colour}
- Each new communicator (sub-group) comprises those processes that specify its value in \textit{colour}
- These processes are assigned new identifiers (ranks, starting at zero) with the order determined by the value of \textit{key} (or by their ranks in the old communicator in event of ties)