Parallel Programming
MPI

Pham Quang Dung

Hanoi, 2012

Outline

1 Introduction

2 Installation

3 MPI routines

Introduction

- Message passing model
- Distributed memory architecture
- Communication is based on **send** and **receive** operations
- Documentation : http ://www.open-mpi.org/doc/

Outline

1 Introduction

2 Installation

3 MPI routines

Installation

- Download openmpi-1.6.2.tar.gz from http ://www.open-mpi.org/software/ompi/v1.6/
- Extract and cd openmpi-1.6.2
- Run : ./configure
- Run : sudo make all install
- Configure library : export LD_LIBRARY_PATH=$LD_LIBRARY_PATH :/usr/local/lib/

Outline

1. Introduction
2. Installation
3. MPI routines

MPI routines

- 8 cores functions allowing to write an MPI application
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
  - MPI_Isend
  - MPI_Irecv

MPI_Init & MPI_Finalize

- int MPI_Init(int* argc, char***argv)
  - Return MPI_SUCCESS or an error code
  - This function initializes the MPI execution environment
  - This function must be called by each MPI process before any other MPI functions is executed
- int MPI_Finalize()
  - Free any resources
  - Each MPI process must call this function before it exits

MPI_Comm_size

- int MPI_Comm_size(MPI_Comm comm, int* size)
  - IN comm : Communicator
  - OUT size : the number of processes in the communication group
  - Communicator : identifies a process group and defines the communication context. All message tags are unique with respect to a communicator
  - MPI_COMM_WORLD : the processes group includes all processes of a parallel application
  - MPI_Comm_size : returns the number of processes in the group of the given communicator

MPI_Comm_rank

- int MPI_Comm_rank(MPI_Comm comm, int* rank)
  - IN comm : Communicator
  - OUT rank : id of the process in the communication group
  - Communicator : identifies a process group and defines the communication context. All message tags are unique with respect to a communicator
  - MPI_COMM_WORLD : the processes group includes all processes of a parallel application
  - MPI_Comm_rank : returns the id of process in the group of the given communicator

Hello world

```c
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>

int main (int argc, char** argv){
   int rank,size;
   MPI_Init(&argc,&argv); /* starts MPI */
   MPI_Comm_rank(MPI_COMM_WORLD,&rank); /* get current process id */
   MPI_Comm_size(MPI_COMM_WORLD,&size); /* get number of processes
       */
   printf("Hello world from process %d of %d\n",rank,size);
   sleep(10);
   MPI_Finalize();
   return 0;
}
```

Hello world

- Compile : mpic++ -o helloworld helloworld.cpp
- Run : mpirun -np 4 helloworld
- Run with hosts file : mpirun –hostfile myhosts.txt -np 4 helloworld
  - myhosts.txt is a text file
  - each line is an IP address of a host in the system

Hello world

```
Hello world from process 0 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
```

MPI_Send

- MPI_Send(void* buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
  - IN buf : address of the send buffer
  - IN count : number of items to be sent
  - IN dtype : type of the items
  - IN dest : Receiver id
  - IN tag : message tags
  - IN comm : Communicator
- It is a blocking function : it terminates when the send buffer can be reused
  - either the message was delivered, or
  - the data were copied to a system buffer

MPI_Recv

- MPI_Recv(void* buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status* status)
  - IN buf : address of the receive buffer
  - IN count : number of items to be received
  - IN dtype : type of the items
  - IN dest : Sender id
  - IN tag : message tags
  - IN comm : Communicator
  - OUT status : the status information
- It is a blocking function : it terminates when the message is available in the reciever buffer
- The message must not be larger than the receiver buffer
- The remaining part of the buffer not used for the recieved message will be unchanged

MPI_Recv - properties

- A message to be recieved must match the sender, the tag, and the communicator
- Sender and tag can be specified as wild card : MPI_ANY_SOURCE, MPI_ANY_TAG
- The actual length of the received message can be determined via MPI_Get_count function
  - int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
    - Output : count is the number of received elements

Two processes send and receive an array

```
int main(int argc, char** argv){
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Status stat;
  int s[MAX] = {0,1,2,3,4,5,6,7,8,9};
  int r[MAX];
  MPI_Send(s,10,MPI_INT,1-rank,rank,MPI_COMM_WORLD);
  MPI_Recv(r,10,MPI_INT,1-rank,1-rank,MPI_COMM_WORLD, &stat);
  printf("Process %d received: ",rank);
  for(int i = 0; i < 10; i++)
    printf("%d ",r[i]);
  printf("\n");
  MPI_Finalize();
  return 0;
}
```

Deadlock

```
int main(int argc, char** argv){
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Status stat;
  int s[MAX] = {0,1,2,3,4,5,6,7,8,9};
  int r[MAX];
  MPI_Recv(r,10,MPI_INT,1-rank,rank,MPI_COMM_WORLD, &stat);
  MPI_Send(s,10,MPI_INT,1-rank,1-rank,MPI_COMM_WORLD);
  printf("Process %d received: ",rank);
  for(int i = 0; i < 10; i++)
    printf("%d ",r[i]);
  printf("\n");
  MPI_Finalize();
  return 0;
}
```

Deadlock - avoid with nonblocking functions

```
int main(int argc, char** argv){
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Status stat;
  MPI_Request req_s;
  MPI_Request req_r;
  int s[MAX] = {0,1,2,3,4,5,6,7,8,9};
  int r[MAX];
  MPI_Irecv(r,10,MPI_INT,1-rank,rank,MPI_COMM_WORLD, &req_r);
  MPI_Isend(s,10,MPI_INT,1-rank,1-rank,MPI_COMM_WORLD, &req_s);

  MPI_Wait(&req_r, &stat); // blocking function, finishes when
      message is received

  printf("Process %d received: ",rank);
  for(int i = 0; i < 10; i++)
    printf("%d ",r[i]);
  printf("\n");
  MPI_Finalize();
  return 0;
}
```

MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void*recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status stat)

- Equivalent to the execution of MPI_Send and MPI_Recv in parallel threads
- sendbuf and recvbuf are different buffers

Example - two processes

```
int main(int argc, char** argv){
  int id, sz;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&sz);
  int s[MAX],r[MAX];
  int n = 10;
  MPI_Status stat;
  for(int i = 0; i < n; i++) s[i] = (1-2*id)*i;

  MPI_Sendrecv(s,n,MPI_INT,1-id,id,r,n,MPI_INT,1-id,1-id,
      MPI_COMM_WORLD,&stat);
  printf("Process %d received: ",id);
  for(int i = 0; i < n; i++)
    printf("%d ",r[i]);
  printf("\n");

  MPI_Finalize();
}
```

Example - multiple processes - roundrobin

```
int main(int argc, char** argv){
  int id, sz;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&sz);

  int s[MAX], r[MAX];
  int n = 10;
  MPI_Status stat;

  for(int i = 0; i < n; i++) s[i] = id;

  int dest = id + 1; if(dest >= sz) dest = 0;
  int src = id - 1; if(src < 0) src = sz-1;

  MPI_Sendrecv(s,n,MPI_INT,dest,id,r,n,MPI_INT,src,src,
      MPI_COMM_WORLD,&stat);

  printf("Process %d received: ",id);
  for(int i = 0; i < n; i++) printf("%d ",r[i]);
  printf("\n");
  MPI_Finalize();
}
```

MPI_Bcast(void* sendbuf, int count, MPI_Datatype type, int root, MPI_Comm comm)

- The content of sendbuf of the process **root** is copied to all other processes
- Function type : Blocking

MPI_Bcast(void* sendbuf, int count, MPI_Datatype type, int root, MPI_Comm comm)

```
1  int main(int argc, char** argv){
     int id, sz;
3    MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD,&id);
5    MPI_Comm_size(MPI_COMM_WORLD,&sz);
     int root = atoi(argv[1]);
7    printf("Process %d started, root = %d\n",id,root);
     int s[MAX];
9    int n = 10;

11   MPI_Status stat;
     if(id == root)
13     for(int i = 0; i < n; i++)
         s[i] = i;
15
     MPI_Bcast(s,n,MPI_INT,root,MPI_COMM_WORLD);
17
     printf("Process %d received: ",id);
19   for(int i = 0; i < n; i++)
       printf("%d ",s[i]);
21   printf("\n");
   MPI_Finalize();
```

MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

- Process **root** receives the data in the send buffer of all processes
- The received data is stored in the receive buffer ordered by the process id of the senders
- Note : **recvcount** is the number of items to be received from each process

MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

```
int main(int argc, char** argv){
int id, sz;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
MPI_Comm_size(MPI_COMM_WORLD,&sz);
int root = atoi(argv[1]);
int s[MAX], r[MAX];
int n = 10;
MPI_Status stat;
for(int i = 0; i < n; i++) s[i] = id;
printf("Process %d buffer s: ",id);
for(int i = 0; i < n; i++) printf("%d ",s[i]); printf("\n");
MPI_Barrier(MPI_COMM_WORLD);

MPI_Gather(s,1,MPI_INT,r,1,MPI_INT,root,MPI_COMM_WORLD);

printf("Process %d received: ",id);
for(int i = 0; i < n; i++) printf("%d ",r[i]); printf("\n");
MPI_Finalize();
}
```

MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Run with 5 processes

```
Process 0 buffer s: 0 0 0 0 0 0 0 0 0 0
Process 1 buffer s: 1 1 1 1 1 1 1 1 1 1
Process 2 buffer s: 2 2 2 2 2 2 2 2 2 2
Process 3 buffer s: 3 3 3 3 3 3 3 3 3 3
Process 4 buffer s: 4 4 4 4 4 4 4 4 4 4
Process 1 received: 327932596 32640 4220900 0 35656272 32767 0 0
    327982656 32640
Process 2 received: 243923636 32620 4220900 0 -1196785472 32767 0 0
    243973696 32620
Process 4 received: 1950198452 32607 4220900 0 -396994064 32767 0 0
    1950248512 32607
Process 3 received: 724302516 32588 4220900 0 1209642080 32767 0 0
    724352576 32588
Process 0 received: 0 0 1 1 2 2 3 3 4 4
```

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

- Process **root** sends the data in the **sendbuf** to all other processes in the communicator **comm**
- **sendcount** is the number of items to be sent to each process
- **sendbuf** is divided into chunks of **sendcount** items
- **recvcount** is the number of items to be received for each process

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

```
int main(int argc, char** argv){
  int id, sz;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&sz);
  int root = 1;
  int s[MAX], r[MAX];
  int n = 10;
  int sendcount = 2;
  int recvcount = 2;
  MPI_Status stat;
  if(id == root) for(int i = 0; i < n; i++) s[i] = i;
  MPI_Scatter(s,sendcount,MPI_INT,r,recvcount,MPI_INT,root,
      MPI_COMM_WORLD);
  printf("Process %d received: ",id);
  for(int i = 0; i < recvcount; i++) printf("%d ",r[i]);
  printf("\n");
  MPI_Finalize();
}
```

MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

Run with 5 processes

```
Process 1 received: 2 3
Process 0 received: 0 1
Process 4 received: 8 9
Process 3 received: 6 7
Process 2 received: 4 5
```

MPI_Reduce(void* sendbuf, void* recvbuf, int sendcount, MPI_Datatype sendtype, MPI_Op op, int root, MPI_Comm comm)

- Reduces values of all processes to a single process **root**
- **op** is the operator :
  - MPI_MAX
  - MPI_MIN
  - MPI_SUM
  - MPI_PROD
  - ..

MPI_Reduce(void* sendbuf, void* recvbuf, int sendcount, MPI_Datatype sendtype, MPI_Op op, int root, MPI_Comm comm)

```
1  int main(int argc, char** argv){
   int id, sz;
3  MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&id);
5  MPI_Comm_size(MPI_COMM_WORLD,&sz);

7  int root = 1;
   int n = 2, S[n], S1[n];
9  for(int i = 0; i < n; i++){ S[i] = 0; S1[i] = i*100+id;}

11 MPI_Reduce(&S1,&S,2,MPI_INT,MPI_SUM,root,MPI_COMM_WORLD);
   printf("Process id = %d has S = ",id);
13 for(int i = 0; i < n; i++)
     printf("%d ",S[i]);
15 printf("\n");
   MPI_Finalize();
17 }
```

MPI_Reduce(void* sendbuf, void* recvbuf, int sendcount, MPI_Datatype sendtype, MPI_Op op, int root, MPI_Comm comm)

Run with 10 processes

```
Process id = 6 has S = 0 0
Process id = 2 has S = 0 0
Process id = 0 has S = 0 0
Process id = 9 has S = 0 0
Process id = 1 has S = 0 0
Process id = 4 has S = 0 0
Process id = 8 has S = 0 0
Process id = 7 has S = 0 0
Process id = 3 has S = 45 1045
Process id = 5 has S = 0 0
```