Electrical Engineering and Computer Science

EECS 358 - INTRODUCTION TO PARALLEL COMPUTING

Lecture 10

Distributed Mem. Message Passing Prog. - I

Outline

- Introduction to message passing
- Message Passing Interface
- Example programs

Distributed Memory Machines

- Each processor has local memory with local address space
- Only way to exchange data is using explicit message passing
- Time taken for message depends on the relative locations of the source and destination processors
- Performance of a parallel program determined by how well the location of data matches its use

Programming Models

- Typically, distributed memory machines are space shared, i.e., users obtain a subpartition of the machine to execute their program
- In some machines subpartitions are also time shared amongst users
- However, such sharing uses gang scheduling, i.e. all the processors in the subpartition are available to the user at the same time
- These factors make the Single Program Multiple Data (SPMD) programming model popular
- In the SPMD model, every processor executes the same program; computation is shared based on processor IDs

Programming Models

- Another programming model for distributed memory machines is the Multiple Program Multiple Data (MPMD) model
- Under this model, processors execute different programs and share the computation
- Most of the current distributed memory operating systems do not support the MPMD model directly
- Have to fake the MPMD model using the SPMD model

Inter Processor Communication

- Processors usually need to communicate data amongst themselves when they share computation
- In local communications, processors communicate with a small set of neighbors; in global communications, processors communicate with many other processors
- In structured communication, communicating processors form a regular structure such as a ring, tree or grid; in unstructured communications communicating processors form arbitrary graphs
- In static communication, the identity of communication partners does not change over time; in contrast, the identity of communication partners in dynamic communications may be determined by data computed at runtime and may be highly variable

Inter Processor Communication

- In synchronous communication, communicating processors execute in a co-ordinated fashion; in contrast, asynchronous communication does not require the communicating processors to co-operate
- Most often used communication paradigm is local, static, structured and sychronous communication
- We use matched SEND-RECV pairs for inter processor communication
- The SEND operation initiates message transmission from the source
- The RECV operation initiates message reception at the destination
- SEND-RECV pairs imply a producer-consumer relationship between processors

Inter Processor Communication

- The SEND/RECV operations can be:
  - Blocking: processor waits until operation has completed
  - Non-blocking: processor does not wait for operation to complete and proceeds with further computation
- Typically SENDs are non-blocking and RECVs are blocking
- The reason for this is that the consumer must wait for the correct data from the producer before using it. On the other hand, the producer can just send the correct data and proceed with other computation
- A SEND/RECV pair used in this manner enforces implicit synchronization between the pair of processors involved

Introduction to Message-Passing Interface (MPI)

- Different programming calls for different message passing machines:
  - Intel iPSC, Paragon - NX System
  - Thinking Machines CM-5 - CMMD System
  - IBM SP1/SP2 - MPL or EUIH System
- MPI - standard for explicit message passing
- Follow-on to portable libraries PVM, P4, EXPRESS, PICL
- Standard needed for:
  - Portability and ease-of-use
  - Providing hardware vendors with a well defined set of routines to implement efficiently
  - Development of the parallel software industry

Introduction to MPI

- MPI provides:
  - Point-to-point message passing
  - Collective communication
  - Support for process groups
  - Support for communication contexts
  - Support for application topologies
  - Enviromental enquiry functions

Basics of MPI

- Although the complete MPI is very complex, you need to know only 6 to get started on writing real applications
- MPI_INIT: Initiate an MPI computation
- MPI_FINALIZE: Terminate an MPI computation
- MPI_COMM_SIZE: Determine number of processes
- MPI_COMM_RANK: Determine my process identifier
- MPI_SEND: Send a message
- MPI_RECV: Receive a message

The MPI_SEND Operation

- The MPI_SEND call syntax:

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

- buf is the address of the send buffer
- count is the number of elements to send (integer)
- datatype is the datatype of send buffer elements (handle)
- dest is the process id of destination process (integer)
- tag is a message tag to identify the message (integer)
- comm identifies a group of processes and a communication context (handle)

The MPI_SEND Operation

- On the initiation of an MPI_SEND, the source processor:
  - Allocates system space for the contents of the buffer using the count argument
  - Copies the contents of the buffer into the system space
  - Uses the tag and dest arguments to record the availability of a message for the destination processor
  - Returns to processing the user program

The MPI_RECV Operation

- The MPI_RECV call syntax:

```
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

- buf is the address of the receive buffer
- count is the number of elements to receive (integer)
- datatype is the datatype of send buffer elements (handle)
- source is the process id of source process (integer)
- tag is a message tag to identify the message (integer)
- comm identifies a group of processes and a communication context (handle)
- status is status of the message. The source, tag and count of the message received can be retrieved from the status.

The MPI_RECV Operation

- On the initiation of an MPI_RECV, the destination processor:
  - Uses the tag and source arguments to check the availability of a message from the source processor
  - If the message has been received, it copies the message into the user's buffer; else it waits until the arrival of the message before copying it into the user's buffer
  - Returns to processing the user program

Useful Auxiliary Primitives

- Obtaining size of partition:

```
MPI_COMM_SIZE(comm, size)
```

  - where comm is the communicator
  - and size is the number of processes in the group comm (integer)

- Obtaining process IDs:

```
MPI_COMM_RANK(comm, pid)
```

  - where comm is the communicator
  - and pid is the process id in the group comm (integer)

Example in C

```
#include "mpi.h"
float a[100];
float b[100];
main()

{
  MPI_INIT();
  MPI_COMM_SIZE(MPI_COMM_WORLD, count);
  MPI_COMM_RANK(MPI_COMM_WORLD, myid);
  for(i=0; i < 100; i++) {
    b[i] = 0;
    a[i] = i;
  }
  if (myid == 0)
    MPI_SEND(&a, 100, MPI_FLOAT, 1, 0, MPI_COMM_WORLD);
  else if (myid == 1)
    MPI_RECV(&b, 100, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, status);
/* here proc 1 has values of b equal to a */
}
  MPI_FINALIZE();
```

Return Status Objects

- The return status object is used after completion of a receive to find the actual length, source, and tag of a message
- Return status object is MPI-defined type and provides information about:
  - The source process for the message
  - The message tag
- The number of elements received is given by:

```
MPI_GET_COUNT(status,datatype,count,ierr)
```

Blocking Behavior

- Blocking send/receive:
  - Returns when send/receive is locally complete
  - Message buffer can be overwritten/read
- Non-blocking send/receive:
  - Returns immediately
  - Message buffer contents cannot be overwritten/read until process verifies completion of send/receive

Non-blocking Send/Recv

- The standard non-blocking send routine:

  `MPI_ISEND(buffer,numitems,datatype,dest,tag,comm,req_id,error)`

  - IN - buffer,numitems,datatype,dest,tag,comm
  - OUT - req_id,error

- The standard non-blocking receive routine:

  `MPI_IRECV(buffer,maxitems,datatype,src,tag,comm,req_id,error)`

  - IN - buffer,maxitems,datatype,src,tag,comm
  - OUT - req_id,error

Flavors of Communication

- For a send operation there are:
  - 4 communication modes: standard, ready, synchronous, buffered
  - 2 blocking modes: blocking, nonblocking
  - $4 \times 2 = 8$ types of send
- For a receive operation there are:
  - 1 communication mode
  - 2 blocking modes
  - $1 \times 2 = 2$ types of receive

Naming Conventions

- Send routines:

Communication Mode	Blocking	Non-blocking
Standard	MPI_SEND	MPI_ISEND
Ready	MPI_RSEND	MPI_IRSEND
Synchronous	MPI_SSEND	MPI_ISSEND
Buffered	MPI_BSEND	MPI_IBSEND

- Receive routines:

Communication Mode	Blocking	Non-blocking
Standard	MPI_RECV	MPI_IRECV

- Any type of receive routine can be used to receive messages from any type of send routine

Send/Receive Operations

- MPI_SENDRECV()
- In many applications, processes send to one process while receiving from another
- Deadlock may arise if care is not taken
- MPI provides routines for such send/receive operations
- For distinct send/receive buffers:

  int 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 *status)

Collective Communication

- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete
- Three broad classes:
  - Synchronization - barrier
  - Data movement routines - broadcast, gather, scatter
  - Global computation routines - reduction, scan

Barrier Routine

- Used to synchronize execution of a group of processes

  `MPI_BARRIER(comm)`

- A barrier is a simple way to separate two phases of computation to ensure that messages in two phases do not interact.

Data Movement Routines

- Broadcast routine implements a one-to-all broadcast where a single named process (root) sends the same data to all other processes.

  `MPI_BCAST(buf, count, type, root, comm)`

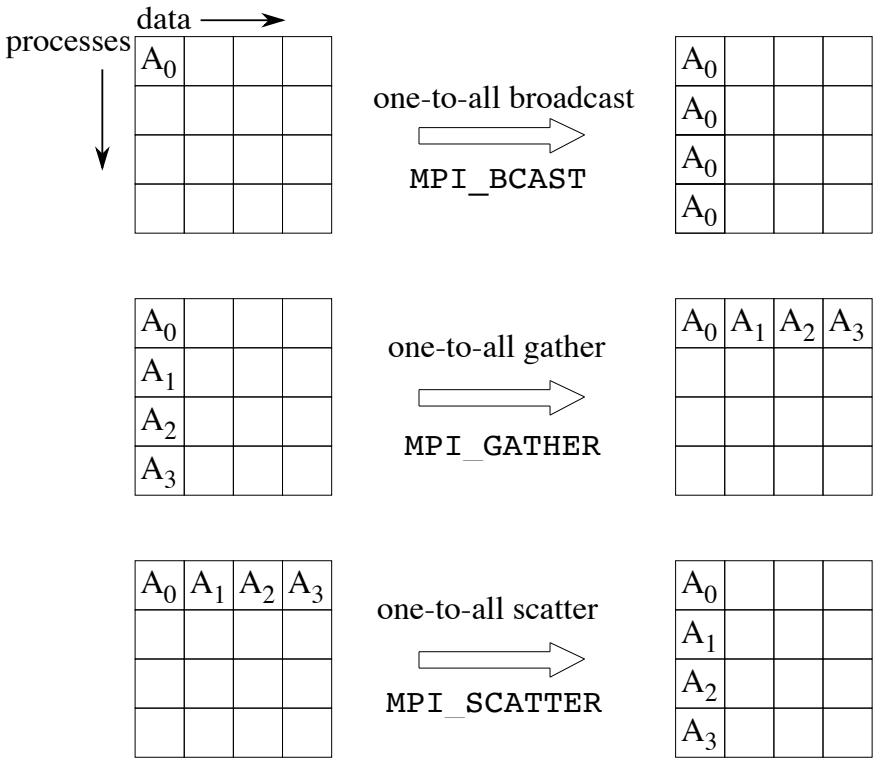
  - buf is the address of the input buffer (output buffer of root)
  - count is the number of elements in input buffer (integer)
  - type is the datatype of input buffer elements (handle)
  - root is the process id of root process (integer)
  - comm identifies a communication context (handle)

Data Movement Routines

```
MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
MPI_SCATTER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

- sendbuf is the address of the input buffer (output buffer of root for scatter)
- sendcount is the number of elements in input buffer (integer)
- sendtype is the datatype of input buffer elements (handle)
- recvbuf is the address of the output buffer
- recvcount is the number of elements in output buffer (integer)
- recvtype is the datatype of output buffer elements (handle)
- root is the process id of root process (integer)
- comm identifies a communication context (handle)

Illustration of MPI Communication Functions



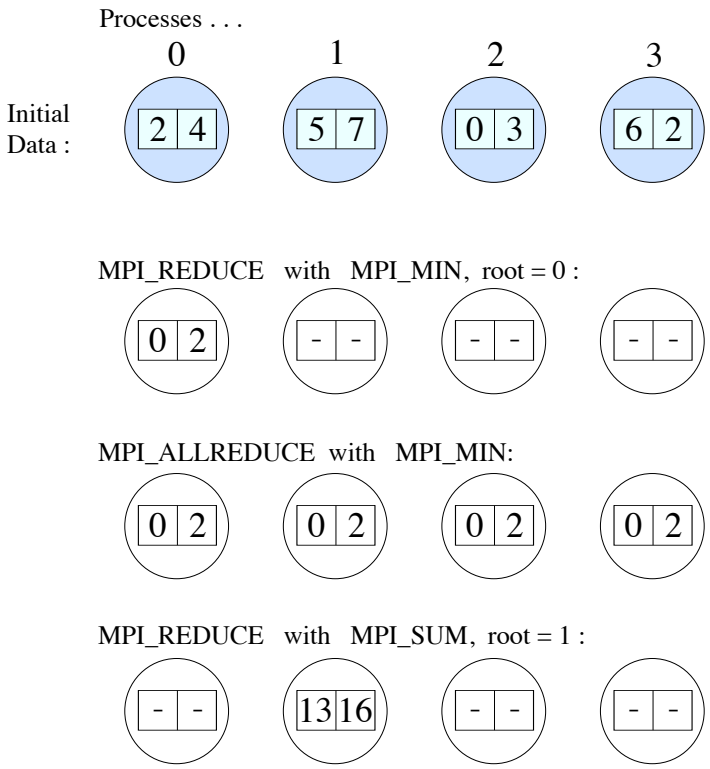
Reduction Operations

- Reduction operations combine the values provided in the input buffer of each process using a specified operation OP, and return combined value into output buffer of single root process or output buffer of all processes.

```
MPI_REDUCE(sendbuf, recvbuf, count, type, op, root, comm)
MPI_ALLREDUCE(sendbuf, recvbuf, count, type, op, comm)
```

  - sendbuf is the address of the input buffer
  - recvbuf is the address of the output buffer
  - count is the number of elements in the send buffer (integer)
  - type is the datatype of buffer elements (handle)
  - op is the operation: MPI_MIN, MPI_MAX, MPI_SUM
  - root is the process id of root process (integer)
  - comm identifies a group of processes and a communication context (handle)

Illustration of Reduction Operations



Example Application: Computation of PI

```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}

int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.14159265358979323846264643;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Example Application: Computation of PI (contd)

```
n = 0;
if (myid == 0)
    {
        printf("Enter the number of intervals: (0 quits) ");
        scanf("%d",&n);
        if (n==0) n=100;
        startwtime = MPI_Wtime();
    }
 MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
 h   = 1.0 / (double) n;
 sum = 0.0;
```

Example Application: Computation of PI (contd)

```
      for (i = myid + 1; i <= n; i += numprocs)
            {
                x = h * ((double)i - 0.5);
                sum += f(x);
            }
       mypi = h * sum;
       MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
       if (myid == 0)
           {
                printf("pi is approximately %.16f, Error is %.16f\n",
                       pi, fabs(pi - PI25DT));
                endwtime = MPI_Wtime();
                printf("wall clock time = %f\n", endwtime-startwtime);
            }
        MPI_Finalize();
}
```

To Run MPI Programs

- Can run MPI on networks of workstations, on for example the IBM SP-2, the IBM J-30, and practically all other parallel machines...
- Compile MPI programs using

```
mpicc -c foo.c
```

- Or use a Makefile (modify makefile in the examples directory)

```
make cpi
```

- Run MPI programs using

```
mpirun -np 4 pi
```

Summary

- Introduction to message passing
- Message Passing Interface
- Example programs
- NEXT LECTURE: Dist. Memory Message-Passing Programming - II
- READING: Foster, "Design and Building of Parallel Programmns", Chapter 8