Introduction to MPI

HY555 Parallel Systems and Grids

Fall 2003

Outline

- MPI layout
- Sending and receiving messages
- Collective communication
- Datatypes
- An example
- Compiling and running

Typical layout of an MPI program

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

int main(int argc,char **argv) {
    int myrank;                        /*Rank of process*/
    int p;                             /*number of processes*/
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    .
    .
    .
    MPI_Finalize();
}
```

Understanding layout

- MPI_Init **MUST** be called before any other MPI functions
  - Sets up the MPI library so it can be used
- After finished with MPI library MPI_Finalize must be called
  - Cleans up unfinished operations
- MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
  - Fills up *myrank* with the rank of a process
  - Each process has a unique rank, starting with 0,1,..
- MPI_Comm_size(MPI_COMM_WORLD,&p);
  - Fills up *p* with the number of available processes

Sending Messages

- int MPI_Send(void *message, int count, MPI_Datatype, int dest, int tag, MPI_Comm comm)

Argument	*Explanation*
message	pointer to the data
count	number of values to be sent
datatype	type of data
dest	destination of the message
tag	type of message
comm	MPI_COMM_WORLD

Receiving messages (1/2)

- int MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Argument	*Explanation*
message	where to store the data
count	how many values to store
datatype	type of data we expect
source	source of the message
tag	type of message
comm	MPI_COMM_WORLD
status	information on the received data

Receiving messages (2/2)

- int MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- *source* can be MPI_ANY_SOURCE
  - We receive messages from any source
  - *status* will contain the rank of process that sent the message
- *tag* can be MPI_ANY_TAG
  - We receive messages with any tag
  - *status* will contain the tag of the message

A simple example (1/2)

- Hello world program (not again!!!)
- We have p processes
- Process with rank 0 will receive a message from each one of the rest p-1 process and will print it

```
#include <stdio.h>
#include “mpi.h”

int main(int argc,char **argv) {
int myrank;                    /* process rank */
int p;                         /* number of processes*/
int source;                    /* source of the message */
int dest;                      /*destination of the message */
int tag=50;                    /* message tag */
char message[100];             /* buffer */
MPI_Status status;

MPI_Init(&argc,&argv);                              /* ! Don’t forget this */
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&p);
```

A simple example (2/2)

```
if(myrank!=0) { /*if I am process other than the one with rank 0*/
     sprintf(message,"Greetings from process %d\n",myrank);
     /*send it to process with rank 0*/
     dest=0;
     /* strlen(message)+1, 1 one stands for '\0' */
     MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,
     MPI_COMM_WORLD);
}
else { /* I am process with rank 0 */
     for(source=1;source<p;source++) {
     /* receive a message from each one of the other p-1 processes
     */MPI_Recv(message,100,MPI_CHAR,source,tag,MPI_COMM_WORL
     D,&status);
     }
}

MPI_Finalize();  /* clean up */
}
```

Collective communication (1/3)

- Broadcast: A single process sends the same data to every process
- int MPI_Bcast(void *message, int count, MPI_Datatype datatype,int root,MPI_Comm comm)
- *message, count and datatype* arguments same as MPI_Send and MPI_Recv
- *root* specifies the process rank that broadcasts the message

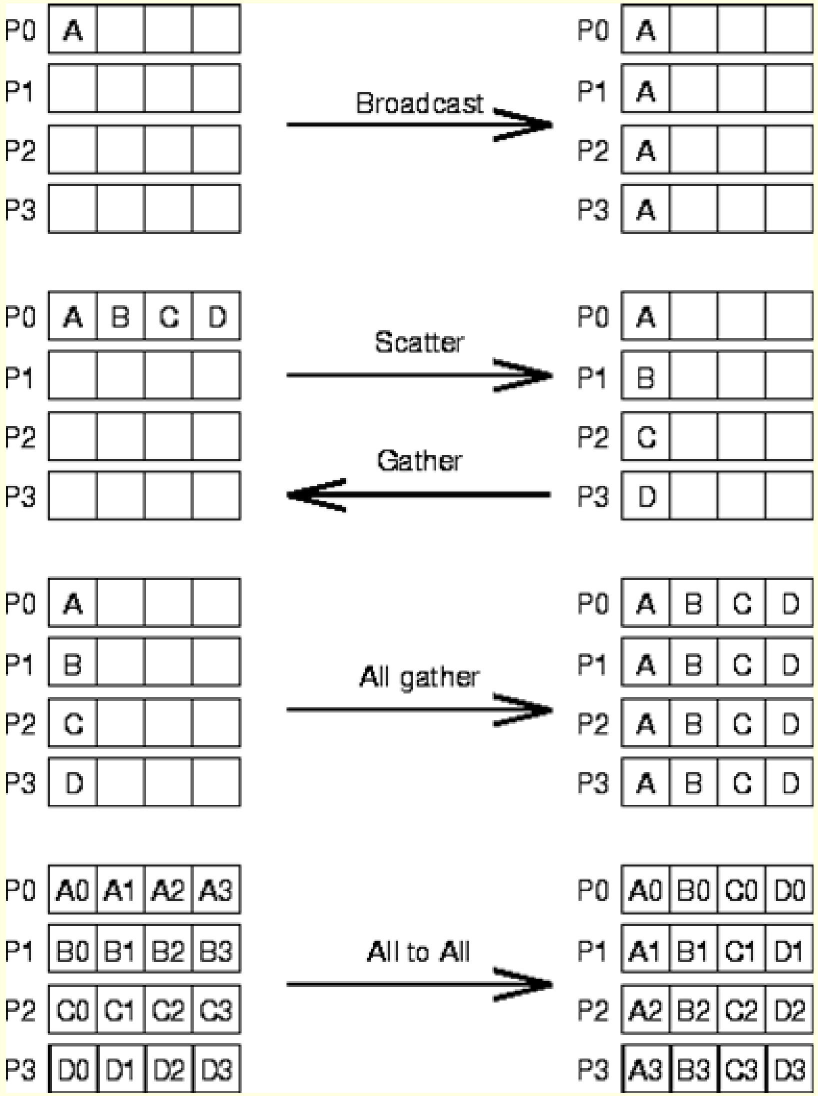
Collective Communication (2/3)

- Reduce: "global" operations
- int MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- Combines the operands stored in *operand using operation op
- Stores the result in *result on process root
- Must be called by ALL process inside comm
  - count,datatype and op must be the same

Collective communication (3/3)

- Synchronizing processes: Barriers
  - Each process blocks on the barrier
  - Unblocks when all processes have reached barrier
  - int MPI_Barrier(MPI_Comm comm)
- Gathering data: MPI_Gather
  - int MPI_Gather(void *send_buf, int send_count, MPI_Datatype send_type, void *recv_buf, int recv_count, MPI_Datatype send_type, int root, MPI_Comm comm)
  - process with rank root gathers data and stores them in recv_buf
  - storage inside recv_buf is based on sender's rank

Collective communication overview



MPI Datatypes

- What types of data can we send and receive?

MPI datatype	**C datatype**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

What about structures? (1/2)

- We must create a new datatype
- Suppose we want to send a struct such as:
  - ```
    struct {
      char display[50];
      int maxiter;
      double xmin,ymin;
      double xmax,ymax;
      int width,height;
    }cmdline;
    ```
- MPI_Type_struct builds a new datatype
  - int MPI_Type_struct(int count, int blocklens[], MPI_Aint indices[], MPI_Datatype old_types[], MPI_Datatype *newtype)
    ```

What about structures? (2/2)

```
/* set up 4 blocks */
int blockcounts[4] = {50,1,4,2};
MPI_Datatype types[4];
MPI_Aint displs[4];
MPI_Datatype cmdtype;

/* initialize types and displs with addresses of items */
MPI_Address( &cmdline.display, &displs[0] );
MPI_Address( &cmdline.maxiter, &displs[1] );
MPI_Address( &cmdline.xmin, &displs[2] );
MPI_Address( &cmdline.width, &displs[3] );
types[0] = MPI_CHAR;
types[1] = MPI_INT;
types[2] = MPI_DOUBLE;
types[3] = MPI_INT;

for (i = 3; i >= 0; i--)
        displs[i] -= displs[0];
MPI_Type_struct( 4, blockcounts, displs, types, &cmdtype );
MPI_Type_commit( &cmdtype );
```

Timing your MPI applications

- Standard approach: gettimeofday in the “master” process
- Instead of gettimeofday you can use MPI_Wtime
  - double MPI_Wtime()
  - simpler than gettimeofday, same semantics
  - returns seconds since an arbitrary moment in the past
- Do not measure initialization functions!!
  - e.g no need to measure MPI_Init

Compile and run

- mpicc used for compilation
  - mpicc –O2 -o program mpi_helloworld.c
- Running programs with mpirun
  - mpirun –np 4 –machinefile machines.solaris program arg1…
  - *np*: number of processes
  - *machines.solaris*: configuration file that lists machines on which MPI programs an be run
    - Simple list of machine names e.g
      chaos.csd.uoc.gr

      geras.csd.uoc.gr

- Add */home/lessons2/hy555/mpi/solaris-x86/bin/* to your path
- For man pages add */home/lessons2/hy555/mpi/solaris-x86/man* to your manpath