SuperMe

Parallel Computing Workshop (Day 2)

Dr. Yifeng Zhu

Dr. Bruce Segee

Electrical and Computer Engineering

University of Maine

May, 2009

Give Credit where Credit is Due

The presentation materials are heavily adopted from:

- *Parallel Programming in C with MPI and OpenMP,* by Michael Quinn, McGraw-Hill Higher Education, ISBN: 0072822562, 2004.
- *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, 2nd edition, by Barry Wilkinson and Michael Allen, ISBN:0131405632, Prentice Hall, 2005.
- *Foundations of Multithreaded, Parallel, and Distributed Programming*, by Gregory R. Andrews, ISBN: 0201357526, Addison Wesley, 2000.
- *Introduction to Parallel Computing*, by Ananth Grama, ISBN:0201648652, Addison Wesley, 2003.
- *Parallel Programming with MPI*, by Peter Pacheco, ISBN:1558603395, Morgan Kaufmann, 1996.
- *Distributed Computing: Principles and Applications*, by M. L. Liu, ISBN 0201796449, Addison-Wesley, 2003.
- *Distributed Systems: Principles and paradigms*, by Andrew S. Tanenbaum & Maarten van Steen, ISBN: ISBN 0-13-088893-1, Prentice-Hall, 2002
- CSI4140 - Introduction to Parallel Computing, Prof. Stefan Dobrev, University of Ottawa
- Parallel Programming with MPI, Ohio Supercomputer Center Presentation Materials

Day 2 Outline

Morning (Lecture):
- **Performance Evaluation**
- **MPI Point-to-point Communication**
- **MPI Collective Communication**
- **MPI Group Communicator**

Afternoon (Lab)
- **MPI Programming in Collective Communication**

Outline

- **Performance Evaluation**
- **MPI Point-to-point Communication**
- **MPI Collective Communication**
- **MPI Group Communicator**

Speedup Factor

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

where t_s is execution time on a single processor and t_p is execution time on a multiprocessor.

***S*(*p*) gives increase in speed by using multiprocessor.**

Use best sequential algorithm with single processor system. Underlying algorithm for parallel implementation might be (and is usually) different.

5

Speedup factor can also be cast in terms of computational steps:

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

Can also extend time complexity to parallel computations.

6

Speedup Example

Example: parallel bubble sort.

- **Bubble sort Ts = 150 seconds.**
- **Tp = 40 seconds**
- **S = 150/40 = 3.75.**

Is this really a fair assessment of the system?

What if serial quicksort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system.

7

Maximum Speedup

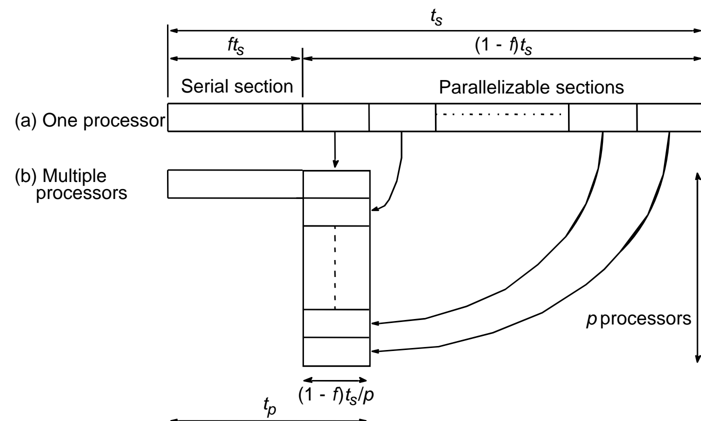
Maximum speedup is usually *p* with *p* processors (linear speedup).

Possible to get superlinear speedup (greater than *p*) but usually a specific reason such as:

- **Extra memory in multiprocessor system**
- **Nondeterministic algorithm**

8

Maximum Speedup
Amdahl's law



Speedup factor is given by:

$$S(p) = \frac{t_s}{ft_s + (1-f)t_s/p} = \frac{p}{1 + (p-1)f}$$

This equation is known as Amdahl's law

Example of Amdahl's Law

10% of a program's code can not be parallelized at all and the remaining 90% can be perfectly parallelized.

a) If we use p=50 processors to run this program in parallel, what would be the maximum speedup that can be obtained ?
b) How many processors should we use in order to obtain an efficiency greater than 0.5 ?

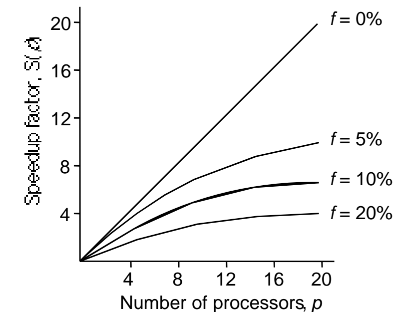
Solution:
----
a) Speed-Up = 1/(f + (1-f)/p) = 1/(0.1 + 0.9/50) = 8.5

b) Efficiency = $\frac{1}{p*(f + (1-f)/p)} > 0.5$

1 / (0.1*p + 0.9) > 0.5

p < 0.55/0.05 ======> p <= 11

Speedup against number of processors

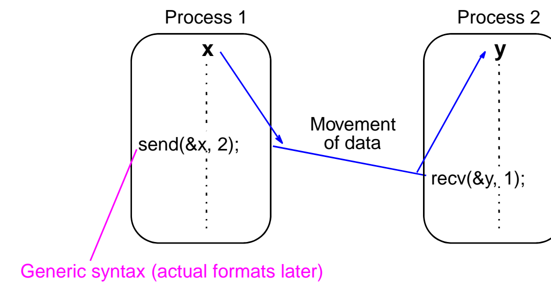


Outline

- Performance Evaluation
- MPI Point-to-point Communication
- MPI Collective Communication
- MPI Group Communicator

Basic "point-to-point" Send and Receive Routines

Passing a message between processes using *send()* and *recv()* library calls:



Synchronous Message Passing

Routines that actually return when message transfer completed.

Synchronous send routine

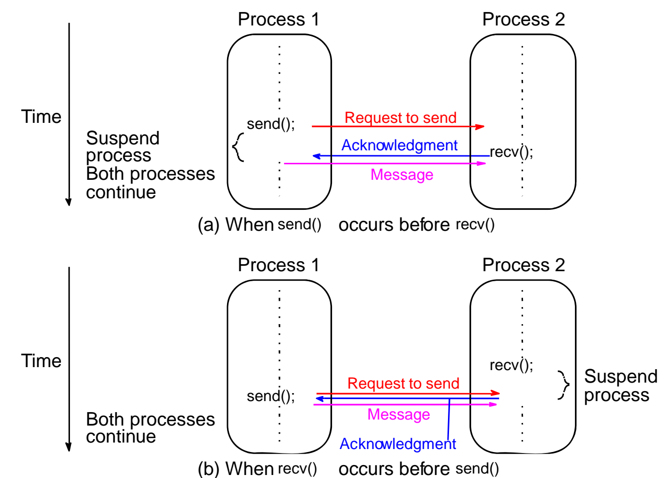
- **Waits until complete message can be accepted by the receiving process before sending the message.**

Synchronous receive routine

- **Waits until the message it is expecting arrives.**

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

Synchronous send() and recv() using 3-way protocol



Asynchronous Message Passing

- Routines that do not wait for actions to complete before returning. Usually require local storage for messages.
- More than one version depending upon the actual semantics for returning.
- In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

17

MPI Definitions of Blocking and Non-Blocking

- **Blocking** - return after their local actions complete, though the message transfer may not have been completed.
- **Non-blocking** - return immediately.

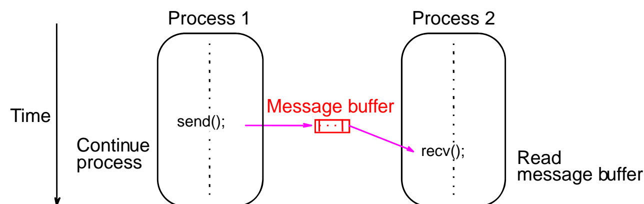
Assumes that data storage used for transfer not modified by subsequent statements prior to being used for transfer, and it is left to the programmer to ensure this.

These terms may have different interpretations in other systems.

18

How message-passing routines return before message transfer completed

Message buffer needed between source and destination to hold message:



19

Asynchronous (blocking) routines changing to synchronous routines

- Once local actions completed and message is safely on its way, sending process can continue with subsequent work.
- Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes re-available - i.e then routine behaves as a synchronous routine.

20

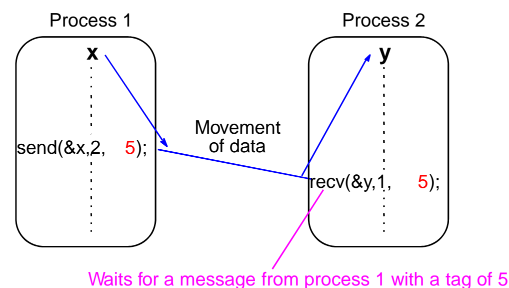
Message Tag

- **Used to differentiate between different types of messages being sent.**
- **Message tag is carried within message.**
- **If special type matching is not required, a wild card message tag is used, so that the recv() will match with any send().**

21

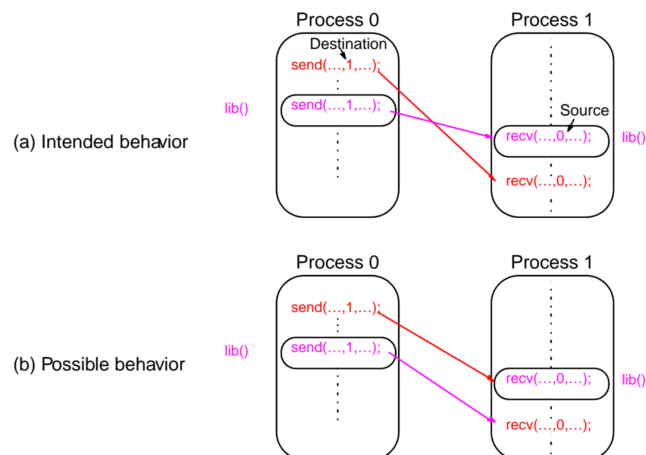
Message Tag Example

- To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y.
- Wild card message tags available



22

Unsafe message passing - Example



23

MPI Basic (Blocking) Send

MPI_SEND(start, count, datatype, dest, tag, comm)

- **The message buffer is described by (`start, count, datatype`).**
- **The target process is specified by `dest`, which is the rank of the target process in the communicator specified by `comm`.**
- **When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.**

24

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

MPI Data Types

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	

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both `source` and `tag`) message is received from the system, and the buffer can be used
- `source` is rank in communicator specified by `comm`, or `MPI_ANY_SOURCE`
- `tag` is a tag to be matched on or `MPI_ANY_TAG`
- receiving fewer than `count` occurrences of `datatype` is OK, but receiving more is an error
- `status` contains further information (e.g. size of message)

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */

if (myrank == 0) {
    int x;
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0,msgtag,MPI_COMM_WORLD,status);
}
```


Example of Send and Receive

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

int main(int argc, char* argv[ ]) {
        int my_rank, p;             // process rank and number of processes
        int source, dest;           // rank of sender and receiving process
        int tag = 0;                // tag for messages
        char mesg[100];             // storage for message
        MPI_Status status;          // stores status for MPI_Recv statements

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        if (my_rank!=0) {
              sprintf(mesg, "Greetings from %d!", my_rank);  // stores into character array
              dest = 0;                          // sets destination for MPI_Send to process 0
              MPI_Send(mesg, strlen(mesg)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
        }                                        // sends string to process 0
        else {
           for(source = 1; source < p; source++){
              MPI_Recv(mesg, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
                                                 // recv from each process
              printf("%s\n", mesg);              // prints out greeting to screen
           }
        }

        MPI_Finalize();                          // shuts down MPI
}
```


MPI Nonblocking Routines

- **Nonblocking send** - **MPI_Isend()** - will return “immediately” even before source location is safe to be altered.
- **Nonblocking receive** - **MPI_Irecv()** - will return even if no message to accept.

Nonblocking Routine Formats

`MPI_Isend(buf,count,datatype,dest,tag,comm,request)`

`MPI_Irecv(buf,count,datatype,source,tag,comm, request)`

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait()` waits until operation completed and returns then.
`MPI_Test()` returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.
Determined by accessing `request` parameter.

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      /* find rank */
if (myrank == 0) {
   int x;
   MPI_Isend(&x,1,MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```


Send Communication Modes (Summary)

- Standard Mode Send - Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering provided, send could complete before receive reached.
- Buffered Mode - Send may start and return before a matching receive. Necessary to specify buffer space via routine *MPI_Buffer_attach*().
- Synchronous Mode - Send and receive can start before each other but can only complete together.
- Ready Mode - Send can only start if matching receive already reached, otherwise error. Use with care.

- Each of the four modes can be applied to both blocking and nonblocking send routines.
- Only the standard mode is available for the blocking and nonblocking receive routines.
- Any type of send routine can be used with any type of receive routine.

Outline

- Performance Evaluation
- MPI Point-to-point Communication
- MPI Collective Communication
- MPI Group Communicator

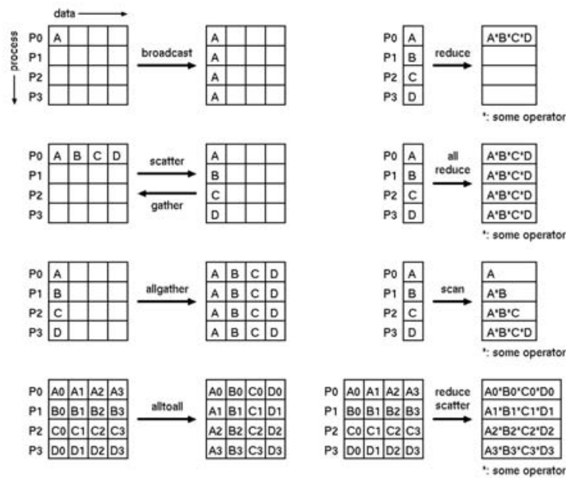
"Group" message passing routines

Have routines that send message(s) to a group of processes or receive message(s) from a group of processes

Higher efficiency than separate point-to-point routines although not absolutely necessary.

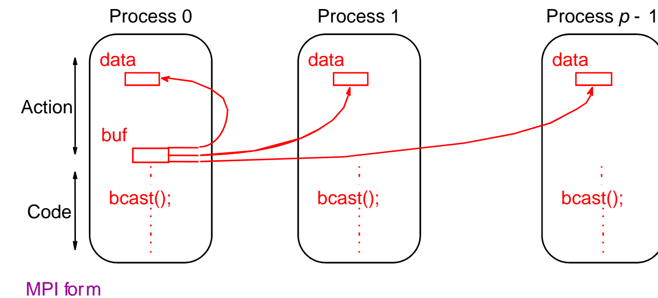
- `MPI_Bcast()` - **Broadcast from root to all other processes**
- `MPI_Gather()` - **Gather values for group of processes**
- `MPI Scatter()` - **Scatters buffer in parts to group of processes**
- `MPI_Alltoall()` - **Sends data from all processes to all processes**
- `MPI_Reduce()` - **Combine values on all processes to single value**
- `MPI_Reduce_scatter()` - **Combine values and scatter results**
- `MPI_Scan()` - **Compute prefix reductions of data on processes**

Big Picture of Group Communication



Broadcast

Sending same message to all processes concerned with problem.
Multicast - sending same message to defined group of processes.



Broadcast

*MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);*

- All processes use the same count, datatype, root, and communicator.
- Before the operation, the root buffer contains a message.
- After the operation, all buffers contain the message from the root process.

Notes:

- root: rank of broadcast root (integer)
- MPI_COMM_WORLD contains a "context" and the "group of all known processes"

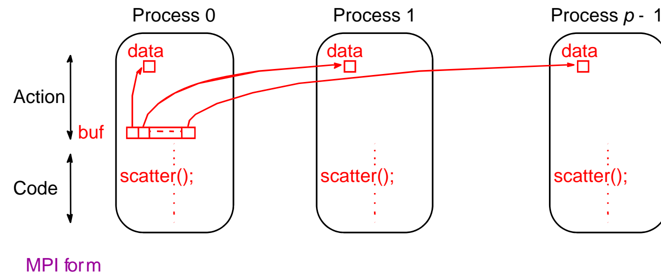
Sample Program: Broadcast

```
#include<mpi.h>
void main (int argc, char *argv[]) {
  int rank;
  double param;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  if(rank==5) param=23.0;
  MPI_Bcast(&param,1,MPI_DOUBLE,5,MPI_COMM_WORLD);
  printf("P:%d after broadcast parameter is
   %f\n",rank,param);
  MPI_Finalize();
}
```

```
P:0 after broadcast parameter is 23.000000
P:6 after broadcast parameter is 23.000000
P:5 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000
P:7 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:4 after broadcast parameter is 23.000000
```

Scatter

Sending each element of an array in root process to a separate process. Contents of *i*th location of array sent to *i*th process.



MPI form

41

Scatter

MPI_Scatter(void *sndbuf, int sndcnt, MPI_Datatype sndtype, void *rcvbuf, int rcvcnt, MPI_Datatype rcvtype, int root, MPI_Comm comm);

- **All processes use the same send and receive counts, datatypes, root and communicator.**
- **Before the operation, the root send buffer contains a message of length `sndcnt * N', where N is the number of processes.**
- **After the operation, the message is divided equally and dispersed to all processes (including the root) following rank order.**

42

Sample Program: Scatter

```
#include <mpi.h>
 void main (int argc, char *argv[]) {
    int rank,size,i,j;
    double param[4],mine;
    int sndcnt,revcnt;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    revcnt=1;
    if(rank==3){
       for(i=0;i<4;i++) param[i]=23.0+i;
       sndcnt=1;
    }
  MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,revcnt,MPI_DOUBL
  E,3,MPI_COMM_WORLD);
    printf("P:%d mine is %f\n",rank,mine);
    MPI_Finalize();
 }
```

```
P:0 mine is 23.000000
P:1 mine is 24.000000
P:2 mine is 25.000000
P:3 mine is 26.000000
```

43

Another Example of Scatter

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv) int argc; char *argv[];  {
        int numtasks, rank, sendcount, recvcount, source;
        float sendbuf[SIZE][SIZE] = {
          {1.0, 2.0, 3.0, 4.0},
          {5.0, 6.0, 7.0, 8.0},
          {9.0, 10.0, 11.0, 12.0},
          {13.0, 14.0, 15.0, 16.0}  };
        float recvbuf[SIZE];

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

        if (numtasks == SIZE) {
            source = 1;
            sendcount = SIZE;
            recvcount = SIZE;
            MPI_Scatter(sendbuf, sendcount, MPI_FLOAT, recvbuf, recvcount, MPI_FLOAT, source, MPI_COMM_WORLD);
            printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0], recvbuf[1],recvbuf[2],recvbuf[3]);
         } else {
            printf("Must specify %d processors. Terminating.\n",SIZE);
         }
         MPI_Finalize();
}
```

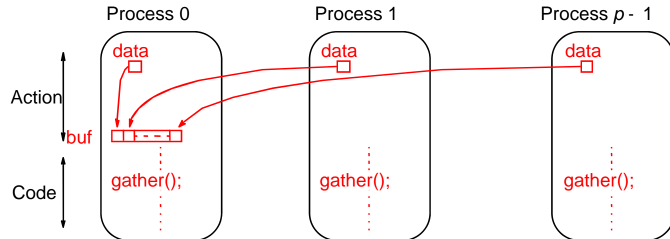
Output:
rank= 0 b= 1.0 2.0 3.0 4.0
rank= 1 b= 5.0 6.0 7.0 8.0
rank= 2 b= 9.0 10.0 11.0 12.0
rank= 3 b= 13.0 14.0 15.0 16.0

What is the output?

44

Gather

Having one process collect individual values from set of processes.



MPI form

Gather

*MPI_Gather(void *sndbuf, int sndcnt, MPI_Datatype sndtype, void *rcvbuf, int rcvcnt, MPI_Datatype rcvtype, int root, MPI_Comm comm);*

- **All processes use the same send and receive counts, datatypes, root and communicator.**
- **This routine is the reverse of MPI_Scatter(): after the operation the root process has in its receive buffer the concatenation of the send buffers of all processes (including its own), with a total message length of `rcvcnt * N', where N is the number of processes.**
- **The message is gathered following rank order.**

Example

To gather items from group of processes into process 0, using dynamically allocated memory in root process:

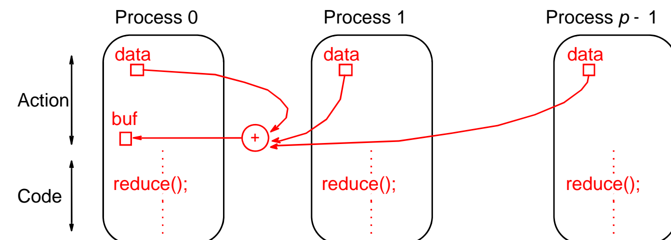
```
/*data to be gathered from processes*/
int data[10];

/* find rank */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
        /*find group size*/
        MPI_Comm_size(MPI_COMM_WORLD, &grp_size);

        /*allocate memory*/
        buf = (int *)malloc(grp size*10*sizeof (int));
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD) ;
```

`MPI_Gather()` **gathers from all processes, including root.**

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example: Values could be gathered and then added together by root:



MPI form

Reduce

*MPI_Reduce(void *sndbuf, void *rcvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);*

- **All processes use the same count, datatype, reduction operation, root and communicator.**
- **After the operation, the root process has in its receive buffer the result of the *pair-wise reduction* of the send buffers of all processes, including its own.**
- **MPI predefines reduction operations, including: MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR.**

Predefined Reduction Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Sample Program: Reduce

```
#include <mpi.h>
/* Run with 16 processes */
void main (int argc, char *argv[]) {
   int rank;
   struct {
     double value;
     int rank;
   } in, out;

   int root;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   in.value=rank+1;
   in.rank=rank;
   root=7;
   MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d max=%lf at rank%d\n",rank,out.value,out.rank);
   MPI_Reduce(&in,&out,1,MPI_DOUBLE_INT,MPI_MINLOC,root,MPI_COMM_WORLD);
   if(rank==root) printf("PE:%d min=%lf at rank %d\n",rank,out.value,out.rank);
   MPI_Finalize();
}
```

```
P:7 max=16.000000 at rank 15
P:7 min=1.000000 at rank 0
```


MINLOC and MAXLOC

- **Designed to compute a global minimum/maximum and and index associated with the extreme value**
  - **Common application: index is the processor rank (see sample program)**
- **If more than one extreme, get the first**
- **Designed to work on operands that consist of a value and index pair**

User-Defined Reduction Operator

```
#include <mpi.h>
 typedef struct {
    double real,imag;
 } complex;

 void cprod(complex *in, complex *inout, int *len,
   MPI_Datatype *dptr) {
    int i;
    complex c;
    for (i=0; i<*len; ++i) {
       c.real=(*in).real * (*inout).real - (*in).imag *
  (*inout).imag;
       c.imag=(*in).real * (*inout).imag + (*in).imag *
  (*inout).real;
       *inout=c;
       in++;
       inout++;
    }
 }

 void main (int argc, char *argv[]) {
    int rank;
    int root;
    complex source,result;
```


User-Defined Reduction Operator

```
    MPI_Op myop;
    MPI_Datatype ctype;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    MPI_Type_contiguous(2,MPI_DOUBLE,&ctype);
    MPI_Type_commit(&ctype);
    MPI_Op_create(cprod,TRUE,&myop);
    root=2;
    source.real=rank+1;
    source.imag=rank+2;

  MPI_Reduce(&source,&result,1,ctype,myop,root,MPI_COMM_WORLD);
    if(rank==root) printf ("PE:%d result is %lf + %lfi\n",rank,
  result.real, result.imag);
    MPI_Finalize();
 }
```

```
P:2 result is -185.000000 + -180.000000i
```


Barrier Synchronization

- **MPI_BARRIER is done in software and can incur a substantial overhead on some machines. In general, you should only insert barriers when they are needed.**

  int MPI_Barrier (MPI_Comm comm)

Outline

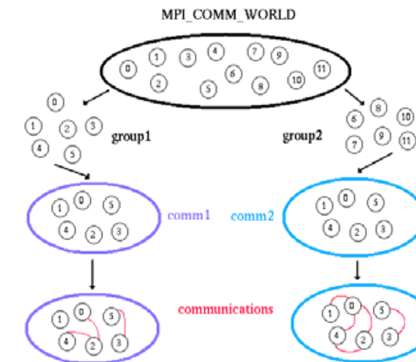
- Performance Evaluation
- MPI Point-to-point Communication
- MPI Collective Communication
- MPI Group Communicator

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called `MPI_COMM_WORLD`

Group and Communicator Management Routines

- A group is an ordered set of processes, ranked from 0 to N-1



Group and Communicator Management Routines

- In MPI, a group is represented within system memory as an object.
- A communicator encompasses a group of processes that may communicate with each other.
  - For example, the handle for the communicator which comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Group and Communicator Management Routines

- A group is an ordered set of processes, ranked from 0 to N-1
- In MPI, a group is represented within system memory as an object.
- A communicator encompasses a group of processes that may communicate with each other.
  - For example, the handle for the communicator which comprises all tasks is MPI_COMM_WORLD.
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Group and Communicator Management Routines

- Primary purposes of group and communicator objects:
  - Allow you to organize tasks, based upon function, into task groups.
  - Enable Collective Communications operations across a subset of related tasks.
  - Provide basis for implementing user defined virtual topologies
  - Provide for safe communications
- Groups/communicators are dynamic - they can be created and destroyed during program execution.
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator.

Group and Communicator Management Routines

- MPI provides over 40 routines related to groups, communicators, and virtual topologies.
- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

MPI_COMM_GROUP

- Definition of MPI_COMM_GROUP
  - Used to return the group underlying the communicator; you get a handle to the group of comm.
- The MPI_COMM_GROUP routine determines the group handle of a communicator.

```
int MPI_Comm_group( MPI_Comm comm, MPI_Group *group )
```

- The function returns an int error flag.

Variable Name	C Type	In/Out	Description
comm	MPI_Comm	Input	Communicator handle
group	MPI_Group *	Output	Group handle

MPI_COMM_GROUP

- Example

```
#include "mpi.h"
MPI_Comm  comm_world;
MPI_Group group_world;

comm_world = MPI_COMM_WORLD;
MPI_Comm_group(comm_world, &group_world);
```

MPI_GROUP_INCL

- **Definition of MPI_GROUP_INCL**
  - Used to form a new group from the processes belonging to group within specified ranks.
- **The MPI_GROUP_INCL routine creates a new group from an existing group and specifies member processes.**

  int MPI_Group_incl(MPI_Group old_group, int count, int *members, MPI_Group *new_group)
- **The function returns an int error flag.**

MPI_GROUP_INCL

int MPI_Group_incl(MPI_Group old_group, int count, int *members, MPI_Group *new_group)

Variable Name	C Type	In/Out	Description
old_group	MPI_Group	Input	Group handle
count	int	Input	Number of processes in new_group
members	int *	Input	Array of size count defining process ranks (in old_group) to be included (in new_group)
new_group	MPI_Group *	Output	Group handle

MPI_COMM_CREATE

- **Definition of MPI_COMM_CREATE**
  - Used to create a new intracommunicator from the processes listed in new_group.
- **The MPI_COMM_CREATE routine creates a new communicator to include specific processes from an existing communicator.**

  int MPI_Comm_create(MPI_Comm old_comm, MPI_Group group, MPI_Comm *new_comm)
- **The function returns an int error flag.**

Variable Name	C Type	In/Out	Description
old_comm	MPI_Comm	Input	Communicator handle
group	MPI_Group	Input	Group handle
new_comm	MPI_Comm *	Output	Communicator handle

Another Example: Group and Communicator Routines

Create two different process groups for separate collective communications exchange. Requires creating new communicators also.

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8
int main(argc,argv) int argc; char *argv[];  {
        int  rank, new_rank, sendbuf, recvbuf, numtasks,
        ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
        MPI_Group  orig_group, new_group;
        MPI_Comm   new_comm;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

        if (numtasks != NPROCS) { printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS); MPI_Finalize(); exit(0); }

        sendbuf = rank;

        /* Extract the original group handle */
        MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

        /* Divide tasks into two distinct groups based upon rank */
        if (rank < NPROCS/2) {
                MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
        } else
                MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);

        /* Create new communicator and then perform collective communications */
        MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
        MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

        MPI_Group_rank (new_group, &new_rank);
        printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);
        MPI_Finalize();
}
```

Sample output:

```
rank= 7 newrank= 3 recvbuf= 22
rank= 1 newrank= 1 recvbuf= 6
rank= 3 newrank= 3 recvbuf= 6
rank= 5 newrank= 1 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 6 newrank= 2 recvbuf= 22
```

MPI_COMM_SPLIT

- Definition of MPI_COMM_SPLIT
  - Used to partition old_comm into separate subgroups. It is similar to MPI_COMM_CREATE.
- The MPI_COMM_SPLIT routine forms new communicators from an existing one.
- The input variable color identifies the group while the key variable specifies a group member.

  int MPI_Comm_split(MPI_Comm old_comm, int color, int key, MPI_Comm *new_comm)
- The function returns an int error flag.

Variable Name	C Type	In/Out	Description
old_comm	MPI_Comm	Input	Communicator handle
color	int	Input	Provides process grouping classification; processes with same color in same group
key	int	Input	Within each group (color), "key" provides control for rank designation within group
new_comm	MPI_Comm *	Output	Communicator handle

69

MPI_COMM_CREATE

- **Note:**
  - MPI_COMM_CREATE is a collective communication routine; it must be called by all processes of old_comm and all arguments in the call must be the same for all processes. Otherwise, error results.
  - MPI_COMM_CREATE returns MPI_COMM_NULL to processes that are not in group.
  - Upon completion of its task, the created communicator may be released by calling MPI_COMM_FREE.
- **Definition of MPI_COMM_FREE**
  - Used to free the communicator or deallocate it. Current and pending operations will complete normally. Deallocation occurs only if no other active references to the communicator are present.

70

Example of MPI_COMM_SPLIT

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char **argv)
{
        MPI_Comm new_comm;
        int myrank, size, even, value;

        MPI_Init (&argc, &argv);
        MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
        MPI_Comm_size (MPI_COMM_WORLD, &size);

        /* Are we odd or even? */
        even = ((myrank % 2) == 0);

        /* Split comm into row and column comms */
        MPI_Comm_split(MPI_COMM_WORLD, even, myrank, &new_comm);
        value = myrank;

        MPI_Bcast(&value, 1, MPI_INT, 0, new_comm);
        printf("Rank %d: Got broadcast value of %d\n", myrank, value);
        MPI_Finalize();
        return 0;
}
```

71

Simple C Example

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
main(int argc, char **argv)
{
        int rank, size, myn, i, N;
        double *vector, *myvec, sum, mysum, total;

        MPI_Init(&argc, &argv );
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        /* In the root process read the vector length, initialize
        the vector and determine the sub-vector sizes */
        if (rank == 0) {
                printf("Enter the vector length : ");
                scanf("%d", &N);
                vector = (double *)malloc(sizeof(double) * N);
                for (i = 0, sum = 0; i < N; i++)
                                vector[i] = 1.0;
                myn = N / size;
        }

        /* Broadcast the local vector size */
        MPI_Bcast(&myn, 1, MPI_INT, 0, MPI_COMM_WORLD );
        /* allocate the local vectors in each process */
        myvec = (double *)malloc(sizeof(double)*myn);
        /* Scatter the vector to all the processes */
```

72

Simple C Example (continued)

```
    MPI_Scatter(vector, myn, MPI_DOUBLE, myvec, myn, MPI_DOUBLE, 0, MPI_COMM_WORLD );

    /* Find the sum of all the elements of the local vector */
    for (i = 0, mysum = 0; i < myn; i++)
        mysum += myvec[i];

    /* Find the global sum of the vectors */
    MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );

    /* Multiply the local part of the vector by the global sum */
    for (i = 0; i < myn; i++)
        myvec[i] *= total;

    /* Gather the local vector in the root process */
    MPI_Gather(myvec, myn, MPI_DOUBLE, vector, myn, MPI_DOUBLE, 0, MPI_COMM_WORLD );

    if (rank == 0)
        for (i = 0; i < N; i++)
        printf("[%d] %f\n", rank, vector[i]);

    MPI_Finalize();
    return 0;
}
```