Introduction of MPI

www.see-grid.eu

SEE-GRID-SCI Training Event,
Yerevan, Armenia, 24-25 July 2008

Mikayel Gyurjyan
Institute for Informatics and Automation Problems
National Academy of Sciences of the Republic of Armenia
Mikayel_Gyurjyan@ipia.sci.am

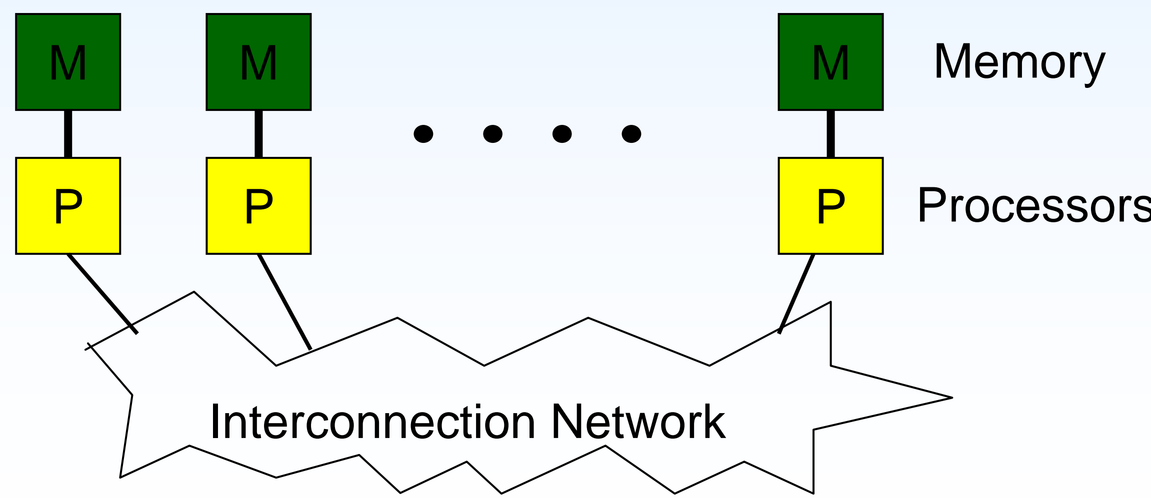
The SEE-GRID-SCI initiative is co-funded by the European Commission under the FP7 Research Infrastructures contract no. 211338

Outline

- Message-Passing Programming Paradigm
- MPI Program Structure
- General MPI functions
- Point-to-Point Communication
- Collective Communication
- Derived Datatypes
- Communicators and Groups

Message-Passing Programming Paradigm

- Each processor in a message-passing program runs a sub-program
  - written in a sequential language
  - all variables are private
  - communicate via special subroutine calls



Creating Parallelism

- Single Program Multiple Data (SPMD)
  - Each MPI process runs a copy of the same program on different data
  - Each copy runs at own rate and is not explicitly synchronized
  - May take different paths through the program
    - Control through *rank* and *number of tasks*

SPMD examples

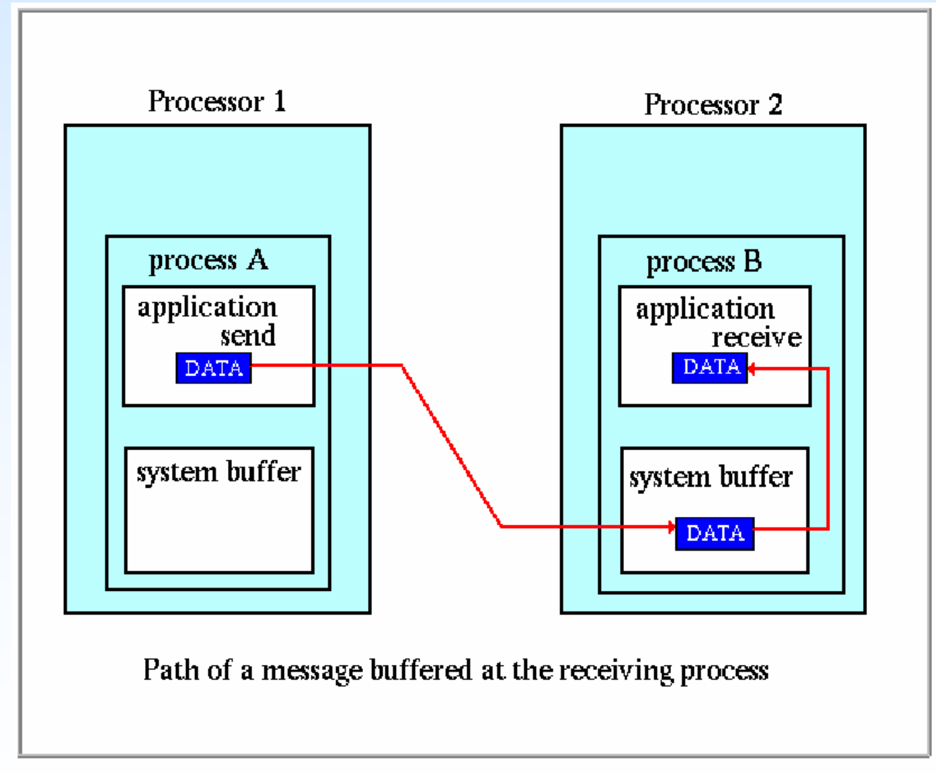
```
main(int argc, char **argv)
{
    if(process is to become Master)
    {
        MasterRoutine(/*arguments*/)
    }
    else /* it is worker process */
    {
        WorkerRoutine(/*arguments*/)
    }
}
```

Creating Parallelism

- Multiple Program Multiple Data
  - Each MPI process can be a separate program
- With OpenMP, pthreads
  - Each MPI process can be explicitly multi-threaded, or threaded via some directive set such as OpenMP

Messages

- Messages are packets of data moving between sub-programs
- The message passing system has to be told the following information
  - Sending processor
  - Source location
  - Data type
  - Data length
  - Receiving processor(s)
  - Destination location
  - Destination size



Path of a message buffered at the receiving process

What Is MPI?

- The Message-Passing Interface (MPI) is a standard for expressing distributed parallelism via message passing.
- MPI consists of a header file, a library of routines and runtime environment.
- Provides a common application programming interface (API) for different platforms.
- Allows C, C++, FORTRAN programmers to write explicit parallel code for distributed systems.

General MPI Program Structure

MPI Include File

Initialize MPI Environment

Do work and perform message communication

Terminate MPI Environment

What does it look like

- Used as function calls in C/C++ and FORTRAN
- In C/C++
  - `#include "mpi.h"`
  - `MPI_Name(...)`
  - Each call (except a few) returns an error code
- In FORTRAN
  - `include 'mpif.h'`
  - `call MPI_Name(...,ierr)`
  - Each call has an extra argument error code `ierr` at the end

Initializing MPI

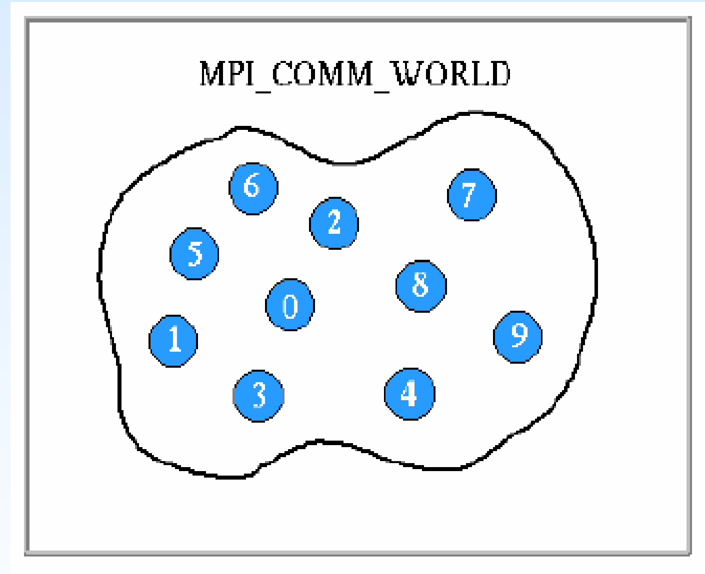
- The first MPI routine called in any MPI program must be **`MPI_Init()`** .
- The C version accepts the arguments to main

```
int MPI_Init(int *argc, char ***argv);
```

- MPI_Init must be called by every MPI program
- Making multiple MPI_Init calls is erroneous

MPI_COMM_WORLD

- MPI_INIT defines a communicator called **MPI_COMM_WORLD** for every process that calls it.
- All MPI communication calls require a communicator argument
- MPI processes can only communicate if they share a communicator.
- A communicator contains a group which is a list of processes
- Each process has it's rank within the communicator
- A process can have several communicators



Communicators

- `MPI_Comm_rank(MPI_comm comm, int *rank)`
  - Returns the rank of the process in comm

- `MPI_Comm_size(MPI_Comm comm, int *size)`
  - Returns the size of the group in comm

Finishing up

- An MPI program should call **MPI_Finalize** when all communications have completed.
- Once called no other MPI calls can be made
- Aborting:

```
MPI_Abort(comm)
```

- Attempts to abort all processes listed in comm if `comm = MPI_COMM_WORLD` the whole program terminates

Managing Processes

- For each program there is a default communicator **MPI_COMM_WORLD**, including all processes
  - All MPI communication calls require a communicator argument.
  - MPI processes can only communicate, if they share a communicator.
  - Each process has it's rank within the communicator
  - A process can have several communicators
- A process calling **MPI_Init()** becomes a member of this communicator.
- To stop participating in the communicator, call **MPI_Finialize()**

Example

C Language

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

int main(argc,argv)
int argc;
char *argv[]; {
int    numtasks, rank, rc;

rc = MPI_Init(&argc,&argv);
if (rc != 0) {
  printf ("Error starting MPI program. Terminating.\n");
  MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);

/*******  do some work *******/

MPI_Finalize();
}
```

Compile and Run Commands

- Compile:
  - shell> mpicc example.c –o example
- Run:
  - shell> mpirun -np 3 -machinefile machines.list example
- The file machines.list contains nodes list:
  - node1
  - node2
  - node3
  - node4
  - node6

Sample Run and Output

- A Run with 3 Processes:
  - shell> mpirun -np 3 -machinefile machines.list example
    - Number of tasks = 3 My rank = 0
    - Number of tasks = 3 My rank = 1
    - Number of tasks = 3 My rank = 2
- A Run by default
  - shell> example
    - Number of tasks = 1 My rank = 0
- Note: Change in process output order. For each run, process mapping can be different. They may run on machines with different load. Hence such difference.

MPI Basic 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	

Point-to-Point Communication

- Simplest form of message passing
- One process sends a message to another
- Several variations on how sending a message can interact with execution of the sub-program

Point-to-Point variations

- Standard mode
  - Blocking - only return from the call when operation has completed
  - Non-blocking - return straight away - can test/wait later for completion
- Non-standard mode
  - Buffered - a buffer must be provided by the application
  - Synchronous - completes only after a matching receive has been posted
  - Ready - may only be called when a matching receive has already been posted

Standard Send : Blocking mode

- **int MPI_Send (void* data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)**
  - **data**: pointer to data
  - **count**: number of elements to be sent
  - **type**: data type
  - **dest**: destination process
  - **tag**: identifying tag
  - **comm**: communicator

Standard Receive : Blocking mode

- **int MPI_Recv (void* data, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Status * status)**
  - **data**: pointer to data
  - **count**: number of elements to be sent
  - **type**: data type
  - **dest**: destination process
  - **tag**: identifying tag
  - **comm**: communicator
  - **status**: sender, tag, and message size

MPI_Probe/MPI_Get_count

- int **MPI_Probe**(int source, int tag, MPI_Comm com, MPI_Status *status)

  - MPI_ANY_SOURCE
  - MPI_ANY_TAG

- int **MPI_Get_count**(MPI_Status *status, MPI_Datatype type, int *count)

hello_world_mpi.c

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

int main (int argc, char* argv[])
{ /* main */
  const int  maximum_message_length = 100;
  const int  master_rank            =   0;
  char       message[maximum_message_length+1];
  MPI_Status status;      /* Info about receive status  */
  int        my_rank;     /* This process ID            */
  int        num_procs;   /* Number of processes in run */
  int        source;      /* Process ID to receive from */
  int        destination; /* Process ID to send to      */
  int        tag = 0;     /* Message ID                 */
  int        mpi_error;   /* Error code for MPI calls   */
   [work goes here]
} /* main */
```

Hello World Startup/Shut Down

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
   [declarations]
  mpi_error = MPI_Init(&argc, &argv);
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  if (my_rank != master_rank) {
     [work of each non-master process]
  } /* if (my_rank != master_rank) */
  else {
     [work of master process]
  } /* if (my_rank != master_rank)...else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Non-master's Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  [MPI startup (MPI_Init etc)]
  if (my_rank != master_rank) {
    sprintf(message, "Greetings from process #%d!“,
        my_rank);
    destination = master_rank;
    mpi_error =
      MPI_Send(message, strlen(message) + 1, MPI_CHAR,
        destination, tag, MPI_COMM_WORLD);
  } /* if (my_rank != master_rank) */
  else {
     [work of master process]
  } /* if (my_rank != master_rank)...else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Master's Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  [MPI startup (MPI_Init etc)]
  if (my_rank != master_rank) {
    sprintf(message, "Greetings from process #%d!“,
        my_rank);
    destination = master_rank;
    mpi_error =
      MPI_Send(message, strlen(message) + 1, MPI_CHAR,
        destination, tag, MPI_COMM_WORLD);
  } /* if (my_rank != master_rank) */
  else {
     [work of master process]
  } /* if (my_rank != master_rank)...else */
  mpi_error = MPI_Finalize();
} /* main */
```

Point-to-Point Communication Modes: Standard Mode -non-blocking

- int **MPI_Isend**(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
- int **MPI_Irecv** (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
  - **buf** - initial address of send buffer
  - **count** - number of elements in send buffer
  - **datatype** - datatype of each send buffer element
  - **dest/source** - rank of destination /source
  - **tag** - message tag
  - **comm-** communicator (handle)
  - **request** - communication request (handle)

MPI_TEST

- int **MPI_Test**(MPI_Request *req, int *flag, MPI_Status *stat)
- int **MPI_Testany**(int count, MPI_Request* array_of_requests, int *index, int *flag, MPI_Status *stat)
- int **MPI_Testall**(int count, MPI_Request* array_of_requests, int *flag, MPI_Status * array_of_stat)
  - **count** - list length
  - **array_of_requests** - array of requests
  - **index** - index of operation that completed, or *MPI_UNDEFINED* if none completed
  - **flag** - true if one of the operations is complete
  - **status** - status object (Status).

MPI_WAIT

- int **MPI_Wait**(MPI_Request *req, MPI_Status *stat)
- int **MPI_Waitany**(int count, MPI_Request *reqs, int *index, MPI_Status *stat)
- int **MPI_Waitall**(int count, MPI_Request *reqs, MPI_Status *array_of_stats)

Point-to-Point Communication Modes: Buffered

- MPI_Bsend
- MPI_Ibsend
- int MPI_Buffer_attach (void *buff, int size)
- int MPI_Buffer_detach (void *buff, int size)
  - Buffered sends do not rely on system buffers
  - The user supplies a buffer that **MUST** be large enough for all messages
  - User need not worry about calls blocking, waiting for system buffer space
  - The buffer is managed by MPI
  - The user **MUST** ensure there is no buffer overflow

Point-to-Point Communication Modes: Synchronous

- MPI_Ssend
- MPI_Isend

  - Can be started (called) at any time.
  - Does not complete until a matching receive has been posted and the receive operation has been started
    - * Does NOT mean the matching receive has completed
  - Can be used in place of sending and receiving acknowledgements
  - Can be more efficient when used appropriately
    - buffering may be avoided

Point-to-Point Communication Modes: Ready Mode

- MPI_Rsend
- MPI_Irsend

  - May ONLY be started (called) if a matching receive has already been posted.
  - If a matching receive has not been posted, the results are undefined
  - May be most efficient when appropriate
    - Removal of handshake operation
  - Should only be used with **extreme** caution

Collective Communication(1)

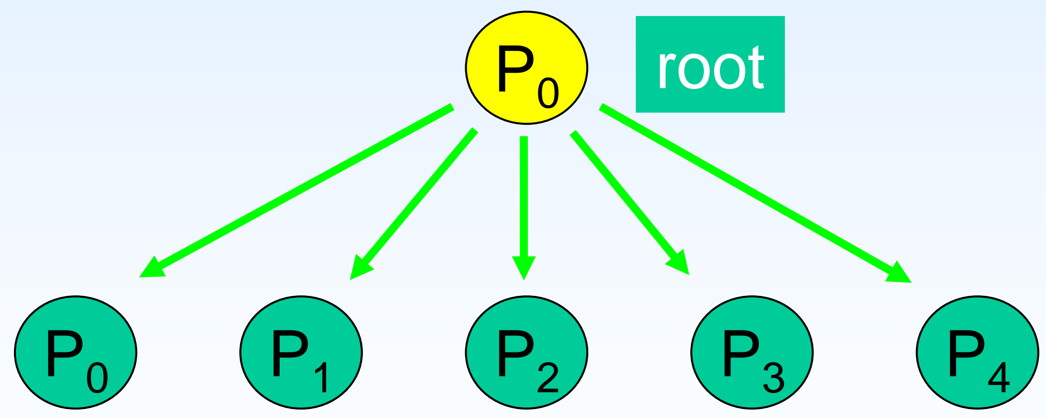
- Amount of data sent must exactly match the amount of data received
- Collective routines are collective across an entire communicator and must be called in the same order from all processors within the communicator
- Collective routines are all blocking
  - This simply means buffers can be re-used upon return
- Collective routines may return as soon as the calling process' participation is complete
  - Does not say anything about the other processors
  - Collective routines may or may not be synchronizing
- No mixing of collective and point-to-point communication

Collective Communication(2)

- Collective communications involve sets of processes.
- Instead of using message tags, communication is coordinated through the use of common variables.
- Examples: broadcast, reduce, scatter/gather, barrier and all-to-all.

Broadcasting(1)

- One process (root) sends a message (data) in ONE operation to all processes in the participating group
- Each of other processes receives the message with the same interface



- Each process has the same copy of data afterwards

Broadcasting(2)

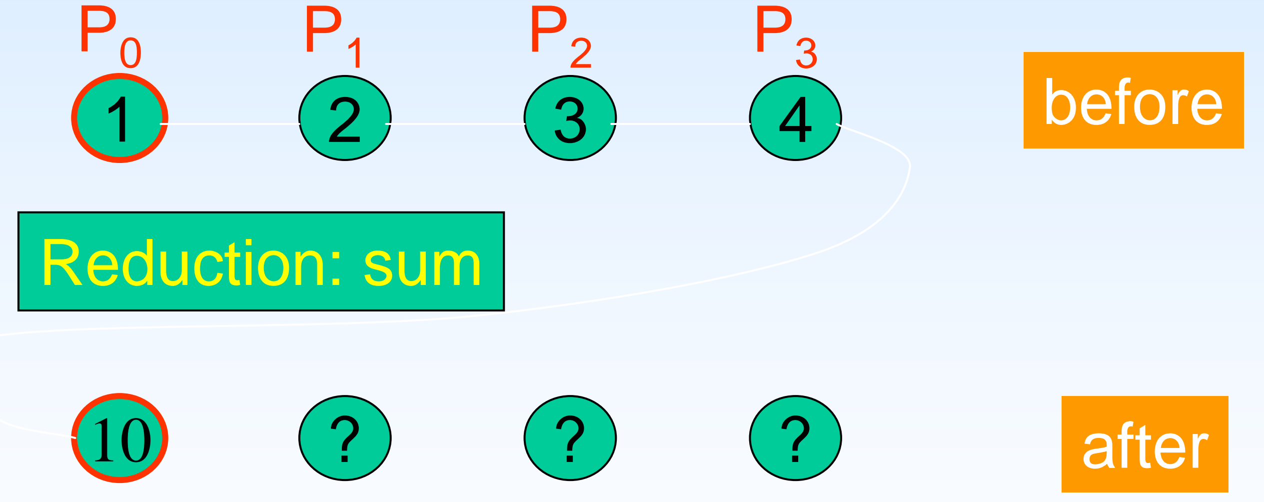
- **MPI_BCAST**(buffer, count, datatype, root, comm)

- [INOUT buffer] - starting address of buffer
  [IN count] - number of entries in buffer
  [IN datatype] - data type of buffer
  [IN root] - rank of broadcast root
  [IN comm] - communicator

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

Reduction(1)

- Collect data, as root, from all processes
- Perform a defined operation on the local data, e.g. sum



- Each one participates, only root has the final results

Reduction(2)

- **MPI_REDUCE**(sendbuf, recvbuf, count, datatype, op, root, comm)
- [IN sendbuf] - address of send buffer
  [OUT recvbuf] -address of receive buffer
  [IN count] - number of elements in send buffer
  [IN datatype] - data type of elements of send buffer
  [IN op] - reduce operation
  [IN root] - rank of root process
  [IN comm] – communicator
- int **MPI_Reduce** (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Example: Reduction with sum

- The call to MPI_Reduce() collects local data (stored in ldata) adds it to get partial sum and stores in gdata

local data

global results

```
MPI_Reduce(ldata, gdata, 1,
MPI_INT,MPI_SUM,0,MPI_COMM_WORLD)
```

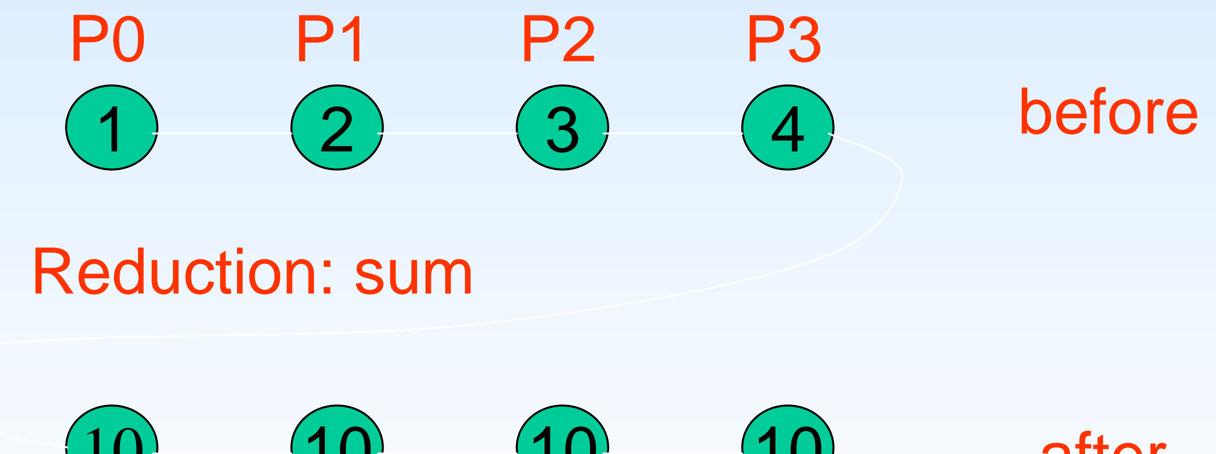
reduction operation

root collects results

- Each process has the same call

All-reduction(1)

- Collect local data across all processes
- Perform a defined operation on local data



- All the processes get a copy of final results

All-Reduction(2)

- **MPI_ALLREDUCE**(sendbuf, recvbuf, count, datatype, op, comm)
- [IN sendbuf] - starting address of send buffer
  [OUT recvbuf] starting address of receive buffer
  [IN count] - number of elements in send buffer
  [IN datatype] - data type of elements of send buffer
  [IN op] - operation
  [IN comm] - communicator
- int **MPI_Allreduce** (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Predifined Operations

- [MPI_MAX] - maximum
  [MPI_MIN] - minimum
  [MPI_SUM] - sum
  [MPI_PROD] - product
  [MPI_LAND] - logical and
  [MPI_BAND] - bit-wise and
  [MPI_LOR] -logical or
  [MPI_BOR] -bit-wise or
  [MPI_LXOR] -logical xor
  [MPI_BXOR] -bit-wise xor

Gather(1)

- Gather – collect data on other processes and stored them in order locally

P1 P2 P3 P_4

7 11 6 49

local

2 | 2 7 11 6 49

receive buffer

P0

Gather(2)

- **MPI_GATHER**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

- [IN sendbuf] - starting address of send buffer
[IN sendcount] -number of elements in send buffer
[IN sendtype] -data type of send buffer elements
[OUT recvbuf] -address of receive buffer (choice, significant only at root)
[IN recvcount] -number of elements for any single receive (integer, significant only at root)
[IN recvtype] -data type of recv buffer elements (significant only at root)
[IN root] - rank of receiving process
[IN comm] - communicator

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

Example: Gathering

- The call to `MPI_Gather()` copies local data in **sbuf** to the receive buffer (**rbuf**) in the root process

local data

```
MPI_Gather(sbuf, sc,MPI_INTEGER,
    rbuf, rc, MPI_INTEGER, 0,
                MPI_COMM_WORLD)
```

receive buffer

- Each process calls, but only the root has all values

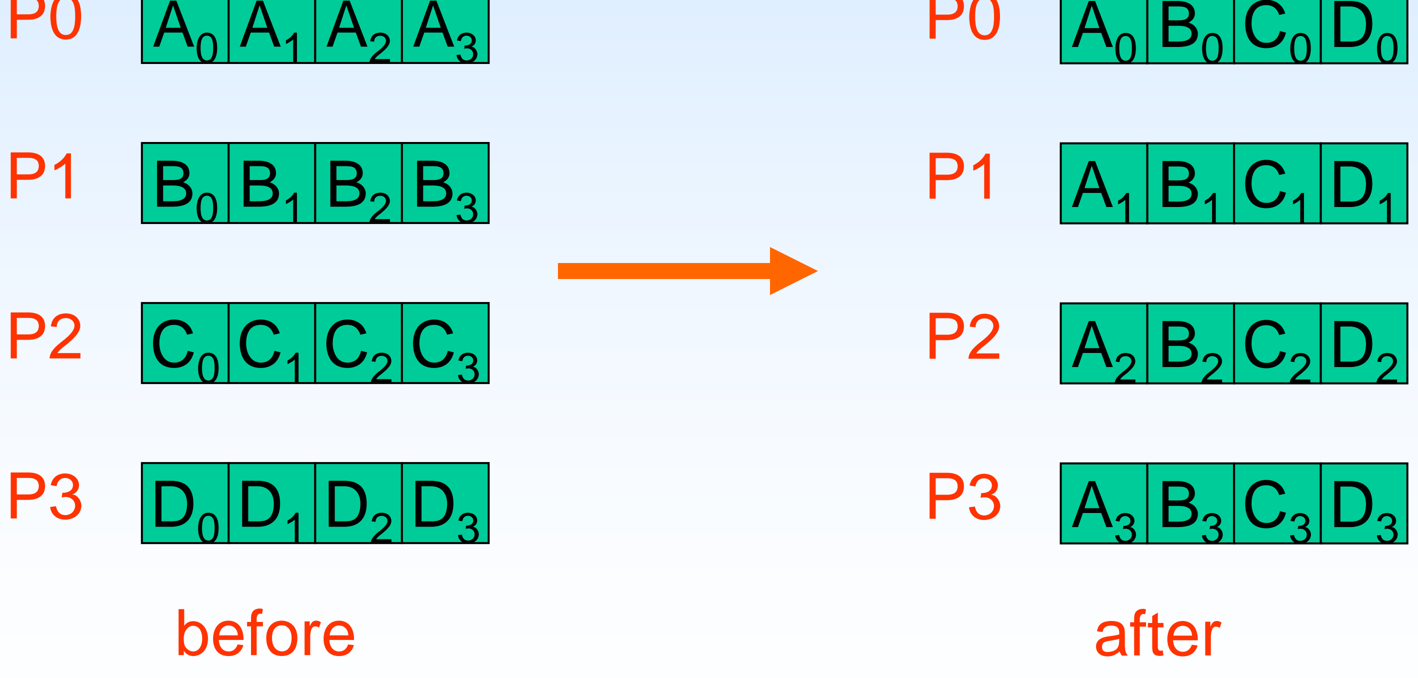
Scatter(1)

- Scatter – A reverse operation of gather, i.e. distribute data to other processes

Scatter(2)

- **MPI_SCATTER**(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- [IN sendbuf] - address of send buffer (significant only at root)
  [IN sendcount] - number of elements sent to each process (significant only at root)
  [IN sendtype] - data type of send buffer elements (significant only at root)
  [OUT recvbuf] - address of receive buffer
  [IN recvcount] - number of elements in receive buffer
  [IN recvtype] - data type of receive buffer elements
  [IN root] - rank of sending process
  [IN comm] - communicator
- int **MPI_Scatter** (void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

All-to-all(1)



All-to-all(2)

- **MPI_ALLTOALL** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

- [IN sendbuf] - starting address of send buffer
  [IN sendcount] - number of elements sent to each process
  [IN sendtype] - data type of send buffer elements
  [OUT recvbuf] - address of receive buffer
  [IN recvcount] - number of elements received from any process
  [IN recvtype] - data type of receive buffer elements
  [IN comm] - communicator

- int **MPI_Alltoall**(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Barrier

- **MPI_BARRIER**(comm)
  [IN comm] - communicator

- int **MPI_Barrier** (MPI_Comm comm)

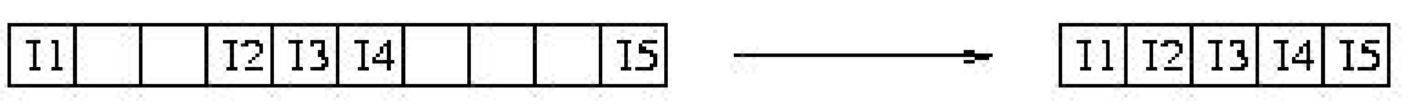
Collective Communication Routines

Collective Communication Routines		
MPI_Allgather	MPI_Allgatherv	MPI_Allreduce
MPI_Alltoall	MPI_Alltoallv	MPI_Barrier
MPI_Bcast	MPI_Gather	MPI_Gatherv
MPI_Op_create	MPI_Op_free	MPI_Reduce
MPI_Reduce_scatter	MPI_Scan	MPI_Scatter
MPI_Scatterv		

Derived Datatypes(1)

- A derived datatype is a sequence of primitive datatypes and displacements
- Derived datatypes are created by building on primitive datatypes
- A derived datatype's *typemap* is the sequence of (primitive type, disp) pairs that defines the derived datatype
  - These displacements need not be positive, unique, or increasing.
- A datatype's type signature is just the sequence of primitive datatypes
- A messages type signature is the type signature of the datatype being sent, repeated *count* times

Derived Datatypes (2)



Typemap =
(MPI_INT, 0) (MPI_INT, 12) (MPI_INT, 16) (MPI_INT, 20) (MPI_INT, 36)

Type Signature =
{MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT}

Type Signature =
{MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT}

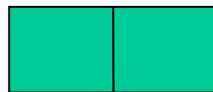
In collective communication, the type signature of data sent must match the type signature of data received!

Derived Type Routines

Derived Types Routines		
MPI_Type_commit	MPI_Type_contiguous	MPI_Type_count
MPI_Type_extent	MPI_Type_free	MPI_Type_hindexed
MPI_Type_hvector	MPI_Type_indexed	MPI_Type_lb
MPI_Type_size	MPI_Type_struct	MPI_Type_ub
MPI_Type_vector		

Datatype Constructors(1)

- MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)
  - Creates a new type representing *count* contiguous occurrences of *oldtype (uses extent(oldtype))*
  - ex: MPI_TYPE_CONTIGUOUS (2, MPI_INT, 2INT)
    - creates a new datatype *2INT* which represents an array of 2 integers



Datatype 2INT

CONTIGUOUS Datatype example

P1 sends 100 integers to P2

P1

```
int buff[100];
MPI_Datatype dtype;
...
...
MPI_Type_contiguous (100,
   MPI_INT, &dtype);
MPI_Type_commit (&dtype);

MPI_Send (buff, 1, dtype, 2, tag,
   MPI_COMM_WORLD)
```

P2

```
int buff[100]

MPI_Recv (buff, 100, MPI_INT, 1, tag,
   MPI_COMM_WORLD, &status)
```

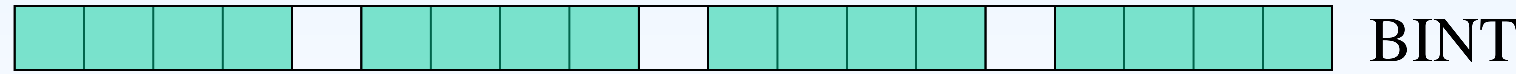
Datatype Constructors (2)

- **MPI_TYPE_VECTOR** (count, blocklength, stride, oldtype, newtype, ierr)
  - Creates a datatype representing *count* regularly spaced occurrences of *blocklength* contiguous *oldtype*s
  - *stride* is in terms of elements of *oldtype (may be negative)*
  - ex: MPI_TYPE_VECTOR (4, 2, 3, 2INT, AINT)

AINT

Datatype Constructors (3)

- MPI_TYPE_CREATE_HVECTOR (count, blocklength, stride, oldtype, newtype, ierr)
  - Identical to MPI_TYPE_VECTOR, except stride is given in bytes rather than elements.
  - ex: MPI_TYPE_CREATE_HVECTOR (4, 2, 20, 2INT, BINT)



BINT

Datatype Constructors (4)

MPI_TYPE_INDEXED (count, blocklengths, displs, oldtype, newtype, ierr)

- Allows specification of non-contiguous data layout
- Good for irregular problems
- ex: MPI_TYPE_INDEXED (3, lengths, displs, 2INT, CINT)
  - lengths = (2, 4, 3)   displs = (0,3,8)

CINT

- Most often, block sizes are all the same (typically 1)
- MPI-2 introduced a new constructor

Pack and Unpack

- MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position, comm, ierr)
- MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype, comm, ierr)
- MPI_PACK_SIZE (incount, datatype, comm, size, ierr)
  - Packed messages must be sent with the type MPI_PACKED
  - Receives must use type MPI_PACKED if the messages are to be unpacked

Communicators and Groups(1)

- Many MPI users are only familiar with MPI_COMM_WORLD
- A communicator can be thought of as a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

Communicators and Groups(2)

- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
- An intracommunicator is used for communication within a single group
- An intercommunicator is used for communication between 2 disjoint groups

Group Management

- You can access information about groups
  - Size, rank within, translate ranks between 2 groups, compare 2 groups
- You can create new groups
  - Get a group associated with a comm
  - Set operations, include or exclude lists

Communicator management

- You can access information about communicators
  - Size of group associated with, rank within group associated with, compare 2
- You can create new communicators
  - Create a new comm for a group
  - Split a group into disjoint set of groups, and create new comms
  - Can create intercommunicators

Process Group Management Routines

Process Group Routines		
MPI_Group_compare	MPI_Group_difference	MPI_Group_excl
MPI_Group_free	MPI_Group_incl	MPI_Group_intersection
MPI_Group_range_excl	MPI_Group_range_incl	MPI_Group_rank
MPI_Group_size	MPI_Group_translate_ranks	MPI_Group_union

Communicators Routines

Communicators Routines		
MPI_Comm_compare	MPI_Comm_create	MPI_Comm_dup
MPI_Comm_free	MPI_Comm_group	MPI_Comm_rank
MPI_Comm_remote_group	MPI_Comm_remote_size	MPI_Comm_size
MPI_Comm_split	MPI_Comm_test_inter	MPI_Intercomm_create
MPI_Intercomm_merge		