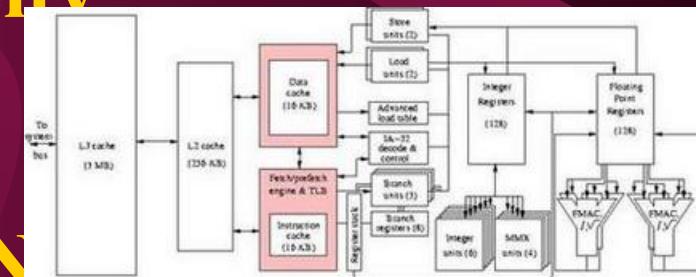
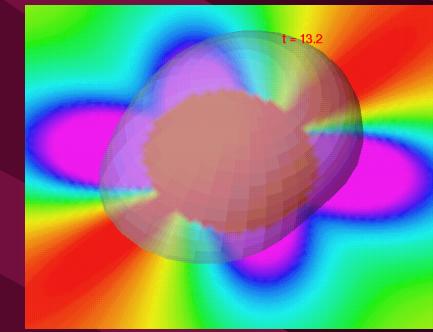
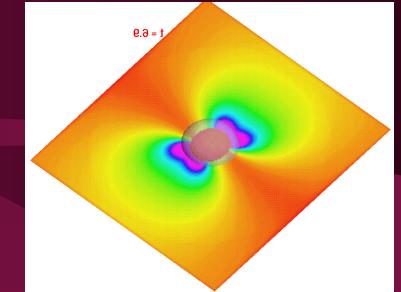
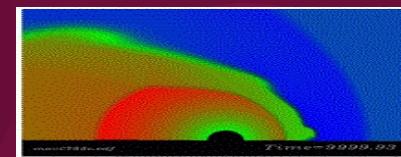


Spring Semester 2005
Geoffrey Fox
Community
Grids Laboratory
Indiana University
505 N Morton
Suite 224
Bloomington IN
gcf@indiana.edu



Abstract of MPI Presentation

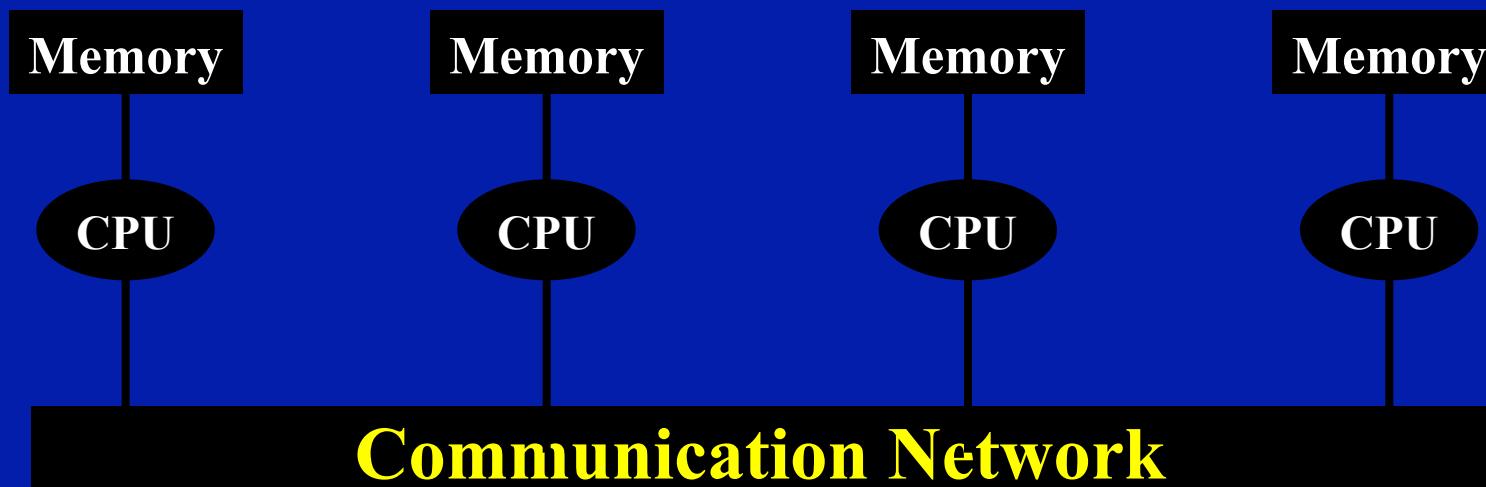
- ◆ This covers MPI from a user's point of view and is to be supplemented by either online tutorials or the recommended book Parallel Programming with MPI, by Peter S. Pacheco, Morgan Kaufmann, 1997.
 - See for example <http://beige.ucs.indiana.edu/I590/>
- ◆ We describe background and history briefly
- ◆ An Overview is based on subset of 6 routines that cover send/receive, environment inquiry (for rank and total number of processors) initialize and finalization with simple examples
- ◆ Processor Groups, Collective Communication and Computation, Topologies, and Derived Datatypes are also discussed

Why and What is Message Passing I

- ◆ We are learning how to perform large scale computations on the world's largest computers
- ◆ These computers are collections of CPU's with various architectures of which one of the most important is distributed memory
- ◆ Each CPU has its own memory and information is communicated between the CPU's by means of explicit messages that take data in one CPU's memory and deliver it to the memory of another CPU
- ◆ Without such messages, the CPU's cannot exchange information and the only way they can be usefully working on the same problem, is by exchanging information.

Why and What is Message Passing II

- ◆ MPI or Message Passing Interface is the agreed standard way of exchanging messages
- ◆ MPI was result of much experimental work from roughly 1983-1993 and is available on essentially all parallel computers.
- ◆ MPI is in the form of a subroutine library for C, Fortran and Java



SPMD Programming

- ◆ In nearly all large scale scientific parallel computing, we use the SPMD (**Single Program Multiple Data**) approach
- ◆ This runs the same code on every processor in the parallel array (This is SP part of SPMD)
- ◆ However each CPU has its own memory and so is processing different data (This is MD part of SPMD)
- ◆ Note we saw this in Hadrian's wall where everybody was building a wall but each bricklayer had a different part of wall
- ◆ As we want to scale to 10,000 to 100,000 separate CPU's we can't put different code on each processor
- ◆ Note code can have data dependent branches, so each processor can actually be executing code specialized to its own situation
- ◆ MPI has the concept of rank so a given processor can find out what part of the wall it is building

Some Key Features of MPI

- ◆ An MPI program defines a set of processes, each executing the same program (**SPMD**)
 - (usually one process per parallel computer node)
- ◆ ... that communicate by calling MPI messaging functions
 - (**point-to-point** and **collective**)
- ◆ ... and can be constructed in a **modular fashion**
 - (**communication contexts** are the key to MPI libraries)
 - Note communicator in MPI specifies both **a context and a process group**
- ◆ Also
 - Support for **Process Groups** -- messaging in subsets of processors
 - Support for **application** dependent (virtual) **topologies**
 - **Inquiry** routines to find out properties of the environment such as number of processors

What is MPI?

- ◆ A standard message-passing library
 - p4, NX, PVM, Express, PARMACS are precursors
- ◆ MPI defines a language-independent interface
 - Not an implementation
- ◆ Bindings are defined for different languages
 - So far, C and Fortran 77, C++ and F90
 - Ad-hoc Java bindings are available
- ◆ Multiple implementations
 - MPICH is a widely-used portable implementation
 - See <http://www.mcs.anl.gov/mpi/>
- ◆ Aimed at High Performance

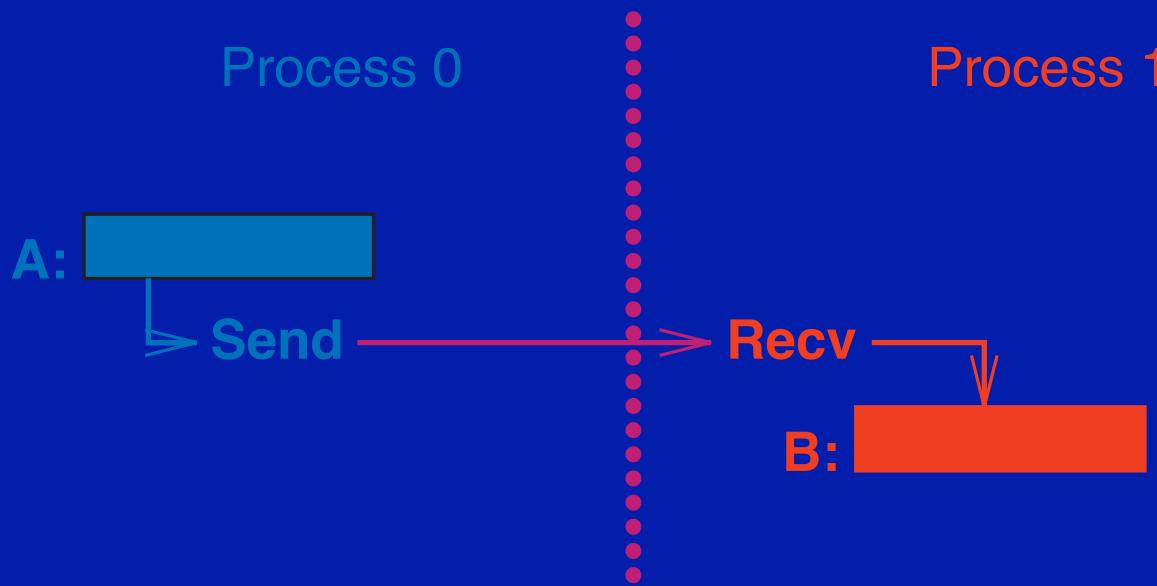
History of MPI

- ◆ Began at Williamsburg Workshop in April 1992
- ◆ Organized at Supercomputing 92 (November 92)
- ◆ Followed format and process similar to those setting Web Standards in W3C and OASIS but less formal
 - Met every 6 weeks for two days
 - Extensive, open email discussions
 - Drafts, readings, votes
- ◆ Pre-final draft distributed at Supercomputing 93
- ◆ Two-month public comment period
- ◆ Final version of draft in May 1994
- ◆ Public and optimized Vendor implementations broadly available with first implementations in 1995
- ◆ MPI-2 agreed in 1998 but major implementation waited until November 2002; it includes parallelism between CPU's and disks (MPI-1 is just between CPU's); more than doubles number of functions in library

Some Difficulties with MPI

- ◆ MPI was designed by the Kitchen Sink approach and has 128 functions and each has many arguments
 - This completeness is strength and weakness!
 - Hard to implement efficiently and hard to learn all its details
 - One can do almost everything with 6 functions
- ◆ It is not a complete operating environment and does not have ability to create and spawn processes etc.
- ◆ PVM is the previous dominant approach
 - It is very simple with much less functionality than MPI
 - However it runs on essentially all machines including heterogeneous workstation clusters
 - Further it is a complete albeit simple operating environment
- ◆ However it is clear that MPI has been adopted as the standard messaging system by parallel computer vendors

Sending/Receiving Messages: Issues

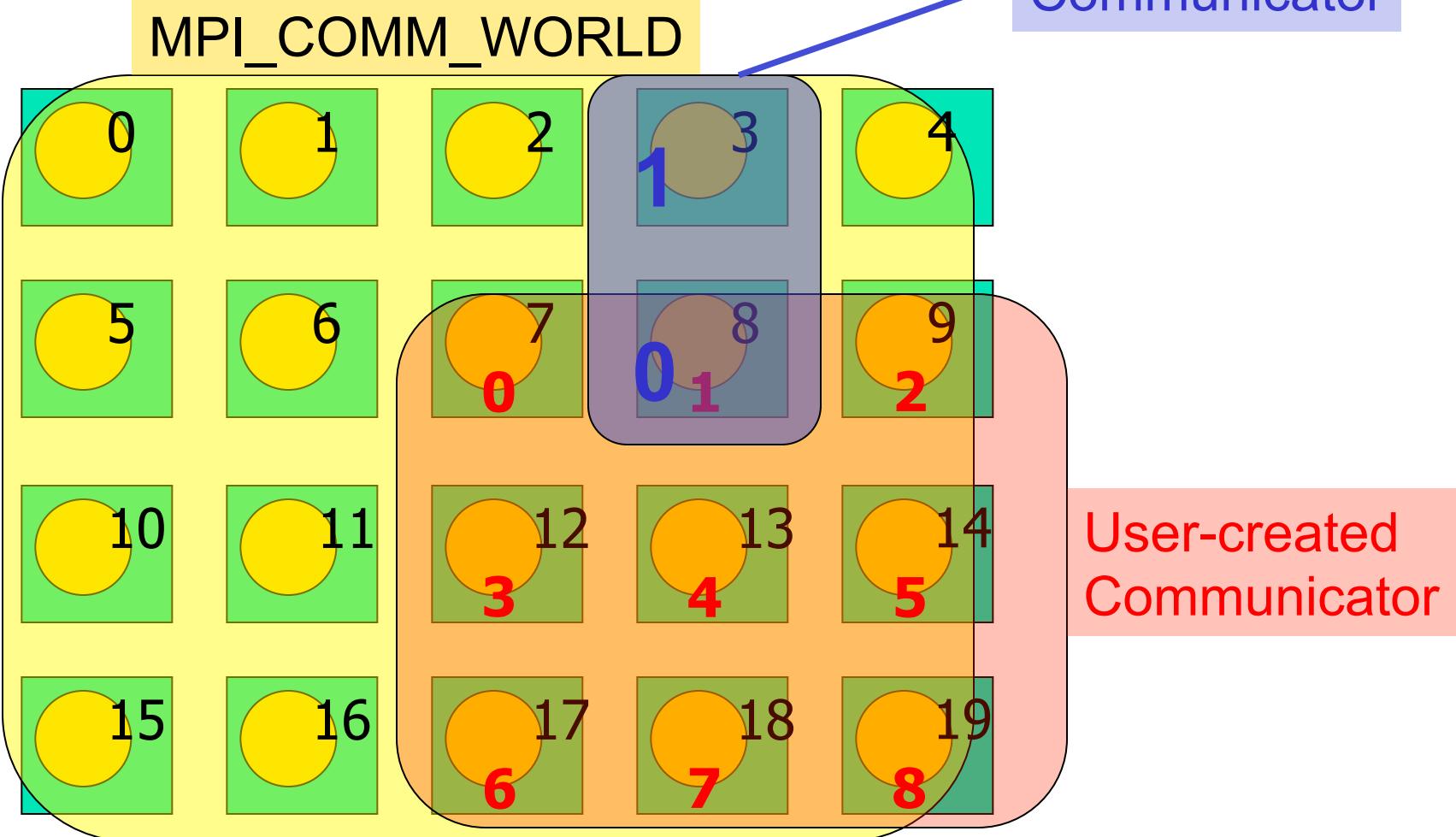


- ◆ **Questions:**
 - What is sent?
 - To whom is the data sent?
 - How does the receiver identify it?
 - Do we acknowledge message?

Key MPI Concepts in a Nutshell

- ◆ MPI uses some concepts we will describe in detail later
- ◆ **Datatype** allows general types of data (including mixtures)
- ◆ **Communicators** allow you to specify sets and subsets of processors involved in this messaging
 - Note MPI supports both point to point and so called **collective communication** involving multiple senders and/or receivers
- ◆ **Tags** allow you to label message so that a receiver can look at it quickly without unpacking buffer and see what message involves and how important it is
- ◆ **Rank** labels the processors and **Size** is total number

MPI Communicators



MPI Conventions

- ◆ All MPI routines are prefixed by MPI_
 - C is always MPI_Xnnnnn(parameters) : C is case sensitive
 - Fortran is case insensitive but we will write MPI_XNNNNNN(parameters)
- ◆ MPI constants are in upper case as are MPI datatypes, e.g. MPI_FLOAT for floating point number in C
- ◆ Specify overall constants with
 - #include "mpi.h" in C programs
 - include "mpif.h" in Fortran
- ◆ C routines are actually integer functions and always return an integer status (error) code
- ◆ Fortran routines are really subroutines and have returned status code as last argument
 - Please check on status codes although this is often skipped!

Standard Constants in MPI

- ◆ There a set of predefined constants in include files for each language and these include:
 - ◆ **MPI_SUCCESS** -- successful return code
 - ◆ **MPI_COMM_WORLD** (everything) and **MPI_COMM_SELF**(current process) are predefined reserved communicators in C and Fortran
- ◆ Fortran elementary **datatypes** are:
 - **MPI_INTEGER**, **MPI_REAL**, **MPI_DOUBLE_PRECISION**, **MPI_COMPLEX**, **MPI_DOUBLE_COMPLEX**, **MPI_LOGICAL**, **MPI_CHARACTER**, **MPI_BYTE**, **MPI_PACKED**
- ◆ C elementary **datatypes** are:
 - **MPI_CHAR**, **MPI_SHORT**, **MPI_INT**, **MPI_LONG**, **MPI_UNSIGNED_CHAR**, **MPI_UNSIGNED_SHORT**, **MPI_UNSIGNED**, **MPI_UNSIGNED_LONG**, **MPI_FLOAT**, **MPI_DOUBLE**, **MPI_LONG_DOUBLE**, **MPI_BYTE**, **MPI_PACKED**

The Six Fundamental MPI routines

- ◆ **MPI_Init** (argc, argv) -- initialize
- ◆ **MPI_Comm_rank** (comm, rank) -- find process label (rank) in group
- ◆ **MPI_Comm_size**(comm, size) -- find total number of processes
- ◆ **MPI_Send** (sndbuf,count,datatype,dest,tag,comm) -- send a message
- ◆ **MPI_Recv**
(recvbuf,count,datatype,source,tag,comm,status) -- receive a message
- ◆ **MPI_Finalize()** -- End Up

MPI_Init -- Environment Management

- ◆ This MUST be called to set up MPI before any other MPI routines may be called
- ◆ For C: int **MPI_Init**(int ***argc**, char ****argv**[])
 - **argc** and **argv[]** are conventional C main routine arguments
 - As usual **MPI_Init** returns an error
- ◆ For Fortran: call **MPI_INIT**(**mpierr**)
 - nonzero (more pedantically values not equal to **MPI_SUCCESS**) values of **mpierr** represent errors

MPI_Comm_rank -- Environment Inquiry

- ◆ This allows you to identify each process by a unique integer called the **rank** which runs from **0** to **N-1** where there are **N** processes
- ◆ If we divide the region 0 to 1 by domain decomposition into **N** parts, the process with **rank r** controls
 - subregion covering **r/N** to **(r+1)/N**
 - for C:int **MPI_Comm_rank(MPI_Comm comm, int *rank)**
 - » **comm** is an MPI communicator of type **MPI_Comm**
 - for FORTRAN: call **MPI_COMM_RANK (comm, rank, mpierr)**

MPI_Comm_size -- Environment Inquiry

- ◆ This returns in integer **size** number of processes in given communicator **comm** (remember this specifies processor group)
- ◆ For C: int **MPI_Comm_size(MPI_Comm comm,int *size)**
- ◆ For Fortran: call **MPI_COMM_SIZE (comm, size, mpierr)**
 - where **comm, size, mpierr** are integers
 - **comm** is input; **size** and **mpierr** returned

MPI_Finalize -- Environment Management

- ◆ Before exiting an MPI application, it is courteous to clean up the MPI state and **MPI_FINALIZE** does this. No MPI routine may be called in a given process after that process has called **MPI_FINALIZE**
- ◆ for C: **int MPI_Finalize()**
- ◆ for Fortran: call **MPI_FINALIZE(MPIERR)**
 - **mpierr** is an integer

Hello World in C plus MPI

- ◆ # all processes execute this program
- ◆ #include <stdio.h>
- ◆ #include <mpi.h>
- ◆ void main(int argc,char *argv[])
- ◆ { int ierror, rank, size
 - MPI_Init(&argc, &argv); # Initialize
 - # In following Find Process Number
 - MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 - if(rank == 0)
 - » printf ("hello World!\n");
 - # In following, Find Total number of processes
 - ierror = MPI_Comm_size(MPI_COMM_WORLD, &size);
 - if(ierror != MPI_SUCCESS)
 - » MPI_Abort(MPI_COMM_WORLD,ierror); # Abort
 - printf("I am processor %d out of total of %d\n", rank, size);
 - MPI_Finalize(); # Finalize }

Hello World with only rank and size.

Processor A

```
rank=0;  
size = 4;  
printf ...;  
printf ...;
```

STDOUT

Processor B

```
rank=1;  
size = 4;  
_____  
printf ...;
```

STDOUT

Processor C

```
rank=2;  
size = 4;  
_____  
printf ...;
```

STDOUT

Processor D

```
rank=3;  
size = 4;  
_____  
printf ...;
```

STDOUT

“Hello World”

I am processor
0 out of 4.

I am processor
1 out of 4.

I am processor
2 out of 4.

I am processor
3 out of 4.

Output merged in some order.

Blocking Send: MPI_Send(C) or MPI_SEND(Fortran)

- ◆ call **MPI_SEND** (

- IN **message** start address of data to send
- IN **message_len** number of items (length in bytes determined by type)
- IN **datatype** type of each data element
- IN **dest_rank** Process number (rank) of destination
- IN **message_tag** tag of message to allow receiver to filter
- IN **communicator** Communicator of both sender and receiver group
- OUT **error_message**) Error Flag (absent in C)

Example MPI_SEND in Fortran

- ◆ **integer count, datatype, dest, tag, comm, mpierr**
- ◆ **real sndbuf(50)**
- ◆ **comm = MPI_COMM_WORLD**
- ◆ **tag = 0**
- ◆ **count = 50**
- ◆ **datatype = MPI_REAL**
- ◆ **call MPI_SEND (sndbuf, count, datatype, dest, tag, comm, mpierr)**

Blocking Receive: MPI_RECV(Fortran)

- ◆ call **MPI_RECV**(
 - IN **start_of_buffer** Address of place to store data(address is Input -- values of data are of course output starting at this address!)
 - IN **buffer_len** Maximum number of items allowed
 - IN **datatype** Type of each data type
 - IN **source_rank** Processor number (rank) of source
 - IN **tag** only accept messages with this tag value
 - IN **communicator** Communicator of both sender and receiver group
 - OUT **return_status** Data structure describing what happened!
 - OUT **error_message**) Error Flag (absent in C)
- ◆ Note that **return_status** is used after completion of receive to find actual received length (**buffer_len** is a maximum length allowed), actual source processor **source_rank** and actual message **tag**
- ◆ We will explain the term “Blocking for SEND/RECV” later – Roughly it means function returns when it is complete

Blocking Receive: MPI_Recv(C)

- ◆ In C syntax is
- ◆ `int error_message = MPI_Recv(`
 - `void *start_of_buffer,`
 - `int buffer_len,`
 - `MPI_DATATYPE datatype,`
 - `int source_rank,`
 - `int tag,`
 - `MPI_Comm communicator,`
 - `MPI_Status *return_status)`

Fortran example: Receive

- ◆ **integer status(MPI_STATUS_SIZE)** An array to store status of received information
- ◆ **integer mpierr, count, datatype, source, tag, comm**
- ◆ **integer recvbuf(100)**
- ◆ **count = 100**
- ◆ **datatype = MPI_REAL**
- ◆ **comm = MPI_COMM_WORLD**
- ◆ **source = MPI_ANY_SOURCE** accept any source processor
- ◆ **tag = MPI_ANY_TAG** accept any message tag
- ◆ call **MPI_RECV (recvbuf, count, datatype, source, tag, comm, status, mpierr)**
 - Note **source** and **tag** can be wild-carded

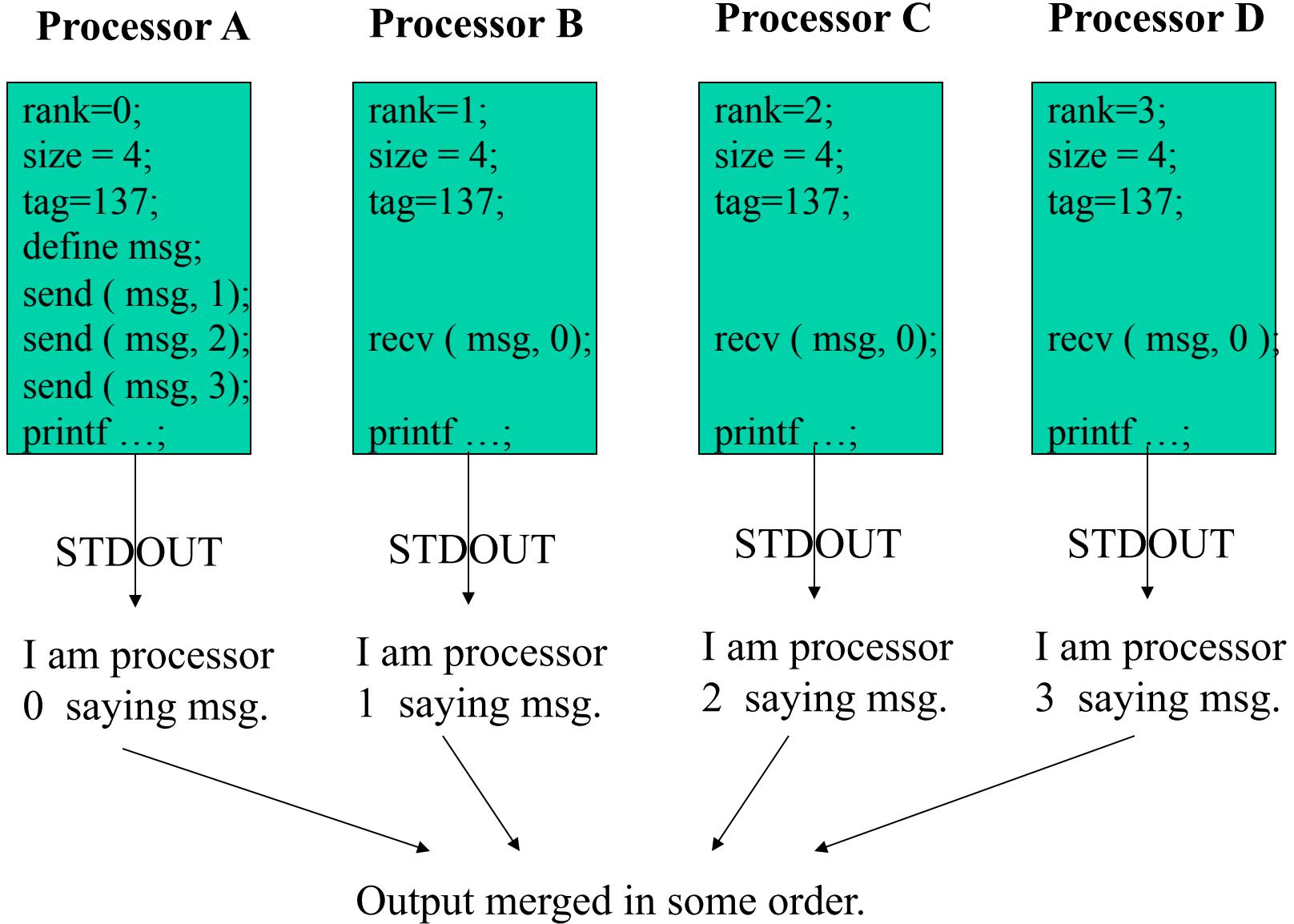
Hello World:C Example of Send and Receive

- ◆ # All processes execute this program
- ◆ #include "mpi.h"
- ◆ main(int argc, char **argv)
- ◆ {
 - char message[20];
 - int i, rank, size, tag=137; # Any value of tag allowed
 - MPI_Status status;
 - MPI_Init (&argc, &argv);
 - MPI_Comm_size(MPI_COMM_WORLD, &size) # Number of Processes
 - MPI_Comm_rank(MPI_COMM_WORLD, &rank); # Who is this process

HelloWorld, continued

- if(**rank == 0**) { # We are on "root" -- Process 0
 - » **strcpy(message,"Hello World");** # Generate message
 - » **for(i=1; i<size; i++)** # Send message to the size-1 other processes
 - » **MPI_Send(message, strlen(message)+1, MPI_CHAR, i, tag, MPI_COMM_WORLD); }**
- else { # Any processor except root -- Process 0
 - » **MPI_Recv(message,20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status); }**
- **printf("I am processor %d saying %s\n", rank, message);**
- **MPI_Finalize();**
- ◆ }

Hello World with send and receive.



Sending and Receiving Arrays

```
#include <mpi.h>
int main(int argc, char **argv) {
    int i, my_rank, nprocs, x[4];
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    if (my_rank == 0) { /* master */
        x[0]=42; x[1]=43; x[2]=44; x[3]=45;
        MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
        for (i=1;i<nprocs;i++)
            MPI_Send(x,4,MPI_INT,i,0,MPI_COMM_WORLD);
    } else { /* worker */
        MPI_Status status;
        MPI_Recv(x,4,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    }
    MPI_Finalize();
    exit(0);
}
```

destination
and
source

user-defined
tag

Max number
of
elements to receive

Can be examined via calls
like MPI_Get_count(), etc.

**Now we go back to MPI
fundamentals and cover in more detail**

Interpretation of Returned Message Status

- ◆ In C **status** is a structure of type **MPI_Status**
 - **status.source** gives actual source process
 - **status.tag** gives the actual message tag
- ◆ In Fortran the **status** is an integer array and different elements give:
 - in **status(MPI_SOURCE)** the actual source process
 - in **status(MPI_TAG)** the actual message tag
- ◆ In C and Fortran, the number of elements (called **count**) in the message can be found from call to
- ◆ **call MPI_GET_COUNT (IN status, IN datatype,
 OUT count, OUT error_message)**
 - whereas in C last argument is missing as returned in function call

Process Groups Tags and Communicators

To Whom It Gets Sent: Process Identifiers

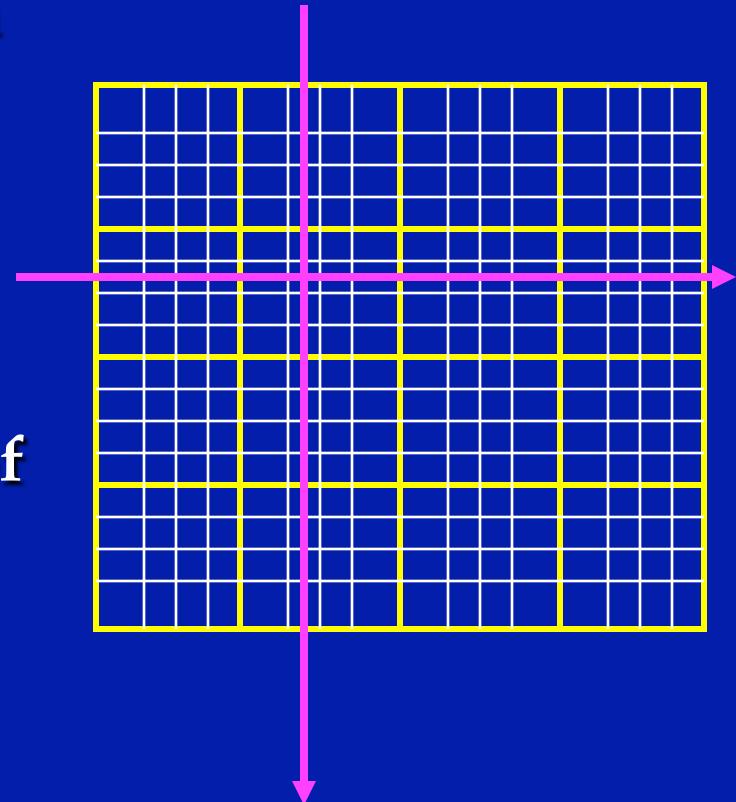
- ◆ 1st generation message passing systems used **hardware addresses**
 - Was **inflexible**
 - » Had to recode on moving to a new machine
 - Was **inconvenient**
 - » Required programmer to map problem topology onto explicit machine connections
 - Was **insufficient**
 - » Didn't support operations over a submachine (e.g., sum across a row of processes)
 - But was **efficient** and quite clear what was happening!

Generalizing the Process Identifier in MPI

- ◆ MPI supports process groups
 - Initial “all” group
 - Group management routines
 - » Split group
 - » Define group from list
- ◆ All communication takes place in groups
 - Source/destination identifications refer to rank in group
 - Communicator = group + context

Why use Process Groups?

- ◆ We find a good example when we consider typical Matrix Algorithm
 - (matrix multiplication)
 - $A_{i,j} = \sum_k B_{i,k} C_{k,j}$
 - summed over k 'th column of B and k 'th row of C
- ◆ Consider a block decomposition of 16 by 16 matrices B and C as for Laplace's equation. (Efficient Decomposition as we study in foilst on matrix multiplication)
- ◆ Each sum operation involves a subset(group) of 4 processors



How Message Is Identified: Message Tags

- ◆ 1st generation message passing systems used an integer “tag” (a.k.a. “type” or “id”) to match messages when received
 - Most systems allowed **wildcard** on receive
 - » wildcard means match any tag i.e. any message
 - » **Unsafe** due to unexpected message arrival
 - Most could match sender **id**, some with **wildcards**
 - » Wildcards unsafe; strict checks inconvenient
 - All systems let **users** pick the **tags**
 - » **Unsafe** for libraries due to interference

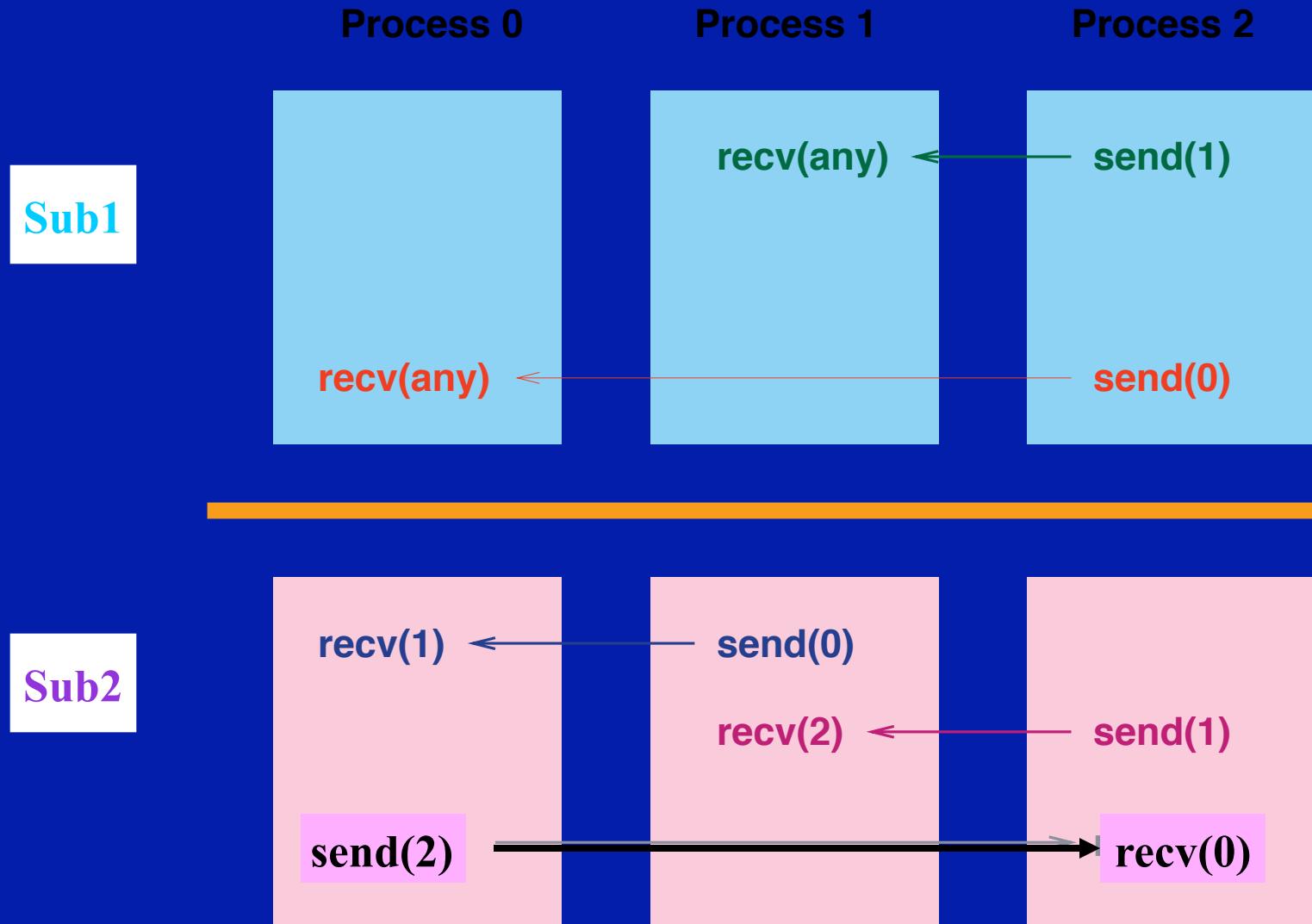
Sample Program using Library

- ◆ Calls **Sub1** and **Sub2** are from different libraries
- ◆ Same sequence of calls on all processes, with no global synch

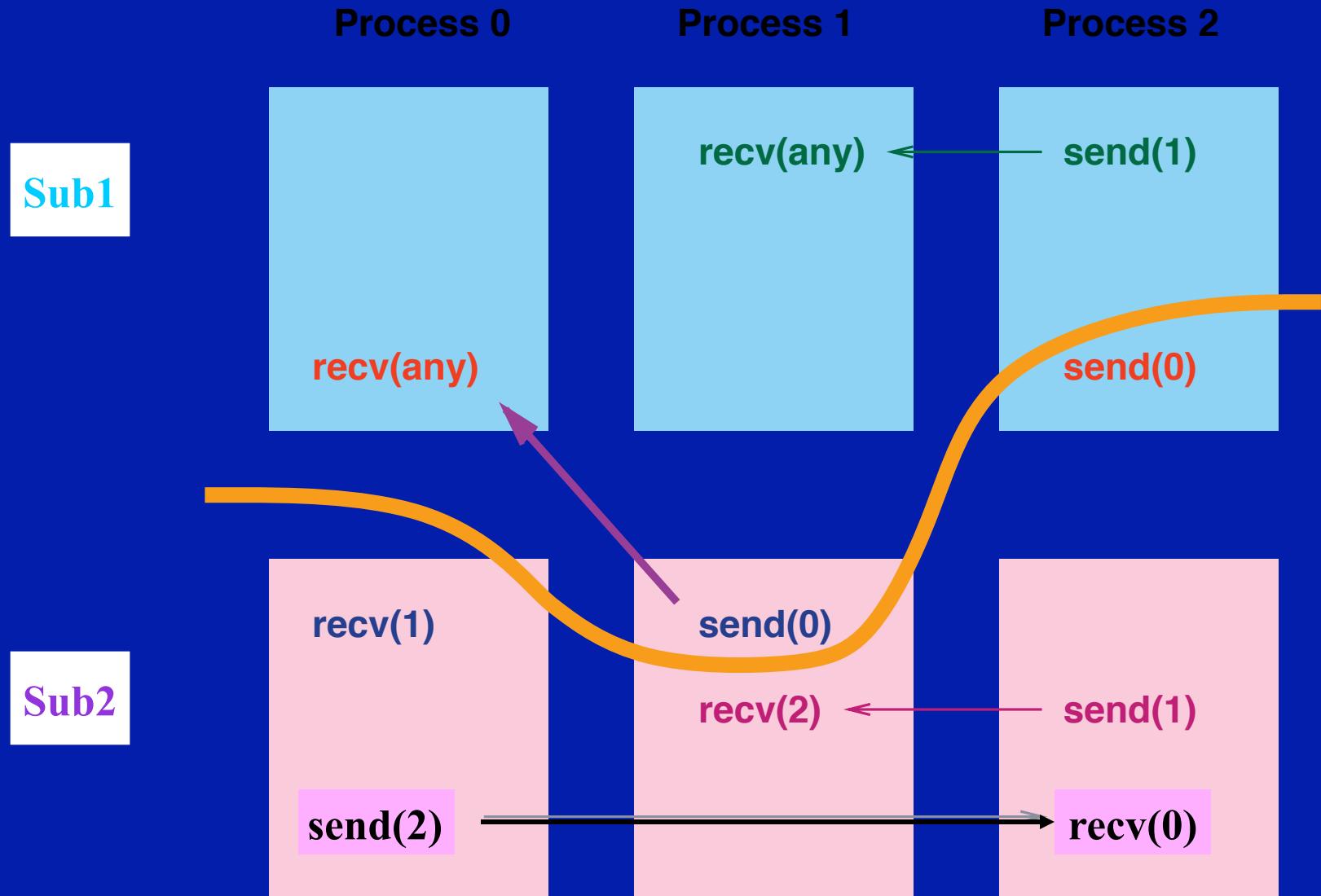
```
Sub1();  
Sub2();
```

- ◆ We follow with two cases showing possibility of error with messages getting mixed up between subroutine calls

Correct Library Execution



Incorrect Library Execution



What Happened?

- ◆ Each library was **self-consistent**
 - Correctly handled all messages it knew about
- ◆ **Interaction** between the libraries killed them
 - “Intercepting” a message broke both
- ◆ The lesson:
 - **Don't take messages from strangers**
- ◆ Other examples teach other lessons:
 - **Clean up your own messages**
 - **Don't use other libraries' tags**
 - **Etc. ...**

Solution to the Tag Problem

- ◆ Generalize **tag** to **tag** and **communicator**
- ◆ A separate communication *context* for each family of messages
 - No wild cards allowed in communicator, for security
 - Communicator allocated by the system, for security
 - Communicator includes groups and possible subsetting for a library within a group – roughly it identifies sets of tasks as a library is usually thought of as a separate task although that's not necessary
- ◆ Tags retained for use **within a context**
 - wild cards OK for tags
- ◆ See
<http://www.llnl.gov/computing/tutorials/workshops/workshop/mpi/MAN.html#Group Management Routines>
for more details on these capabilities

The many Collective Communication Functions

Collective Communication

- ◆ Provides standard interfaces to common global operations
 - Synchronization
 - Communications, i.e. movement of data
 - Collective computation
- ◆ A collective operation uses a process group
 - All processes in group call same operation at (roughly) the same time
 - Groups are constructed “by hand” with MPI group manipulation routines or by using MPI topology-definition routines
- ◆ Message tags not needed (generated internally)
- ◆ All collective operations are blocking.

Some Collective Communication Operations

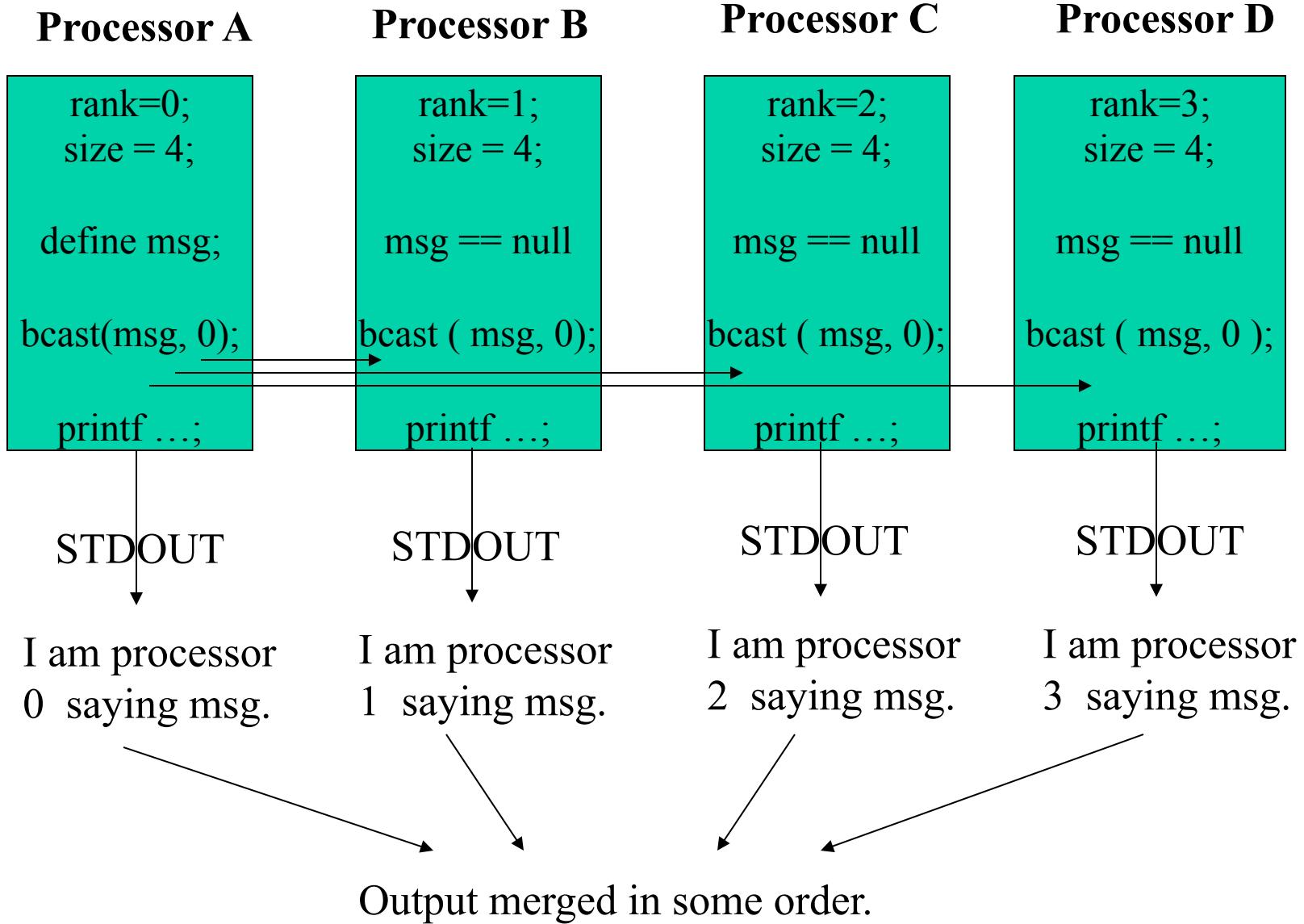
- ◆ **MPI_BARRIER**(comm) Global Synchronization within a given communicator
- ◆ **MPI_BCAST** Global Broadcast
- ◆ **MPI_GATHER** Concatenate data from all processors in a communicator into one process
 - **MPI_ALLGATHER** puts result of concatenation in all processors
- ◆ **MPI_SCATTER** takes data from one processor and scatters over all processors
- ◆ **MPI_ALLTOALL** sends data from all processes to all other processes
- ◆ **MPI_SENDRECV** exchanges data between two processors -- often used to implement "shifts"
 - this viewed as pure point to point by some

Hello World:C Example of Broadcast

- ◆

```
#include "mpi.h"
main( int argc, char **argv )
{
    char message[20];
    int rank;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); # Who is this
processor
    if( rank == 0 )  # We are on "root" -- Processor 0
        » strcpy(message,"Hello MPI World"); # Generate message
    # MPI_Bcast sends from root=0 and receives on all other processor
    MPI_Bcast(message,20, MPI_CHAR, 0, MPI_COMM_WORLD);
    printf("I am processor %d saying %s\n", rank, message);
    MPI_Finalize();
}
```
- ◆ Note that all processes issue the broadcast operation, process 0 sends the message and all processes receive the message.

Hello World with broadcast.



Collective Computation

- ◆ One can often perform computing during a collective communication
- ◆ MPI_REDUCE performs reduction operation of type chosen from
 - maximum(value or value and location), minimum(value or value and location), sum, product, logical and/or/xor, bit-wise and/or/xor
 - e.g. operation labeled MPI_MAX stores in location **result** of processor **rank** the global maximum of original in each processor as in
 - call MPI_REDUCE(original, **result**, 1, MPI_REAL, MPI_MAX, **rank**, comm, ierror)
 - » One can also supply one's own reduction function
- ◆ MPI_ALLREDUCE is same as MPI_REDUCE but it stores result in all -- not just one -- processors
- ◆ MPI_SCAN performs reductions with result for processor **r** depending on data in processors **0** to **r**

Examples of Collective Communication/Computation

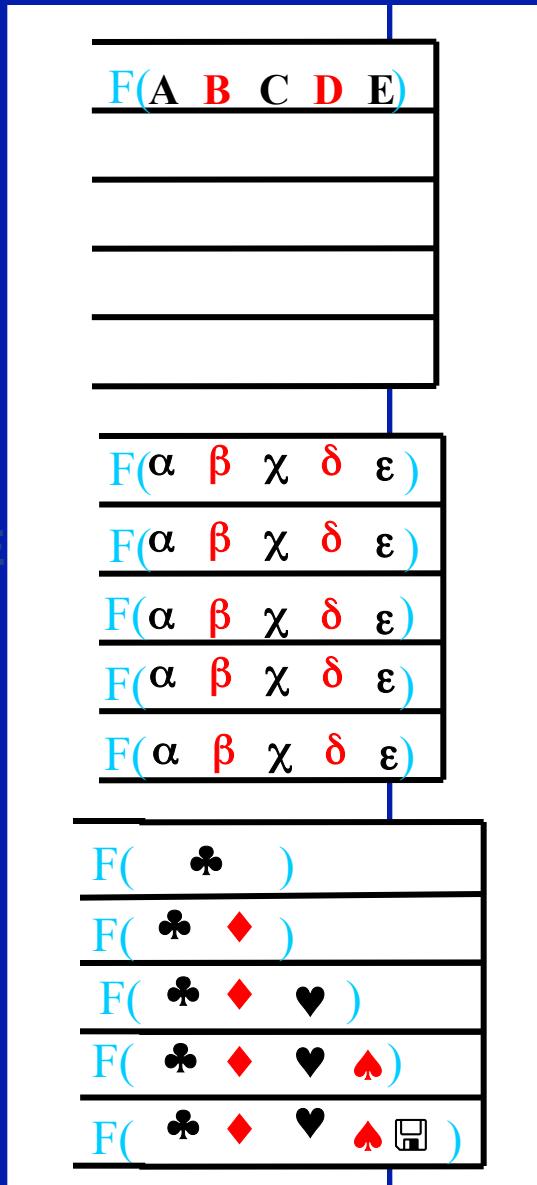
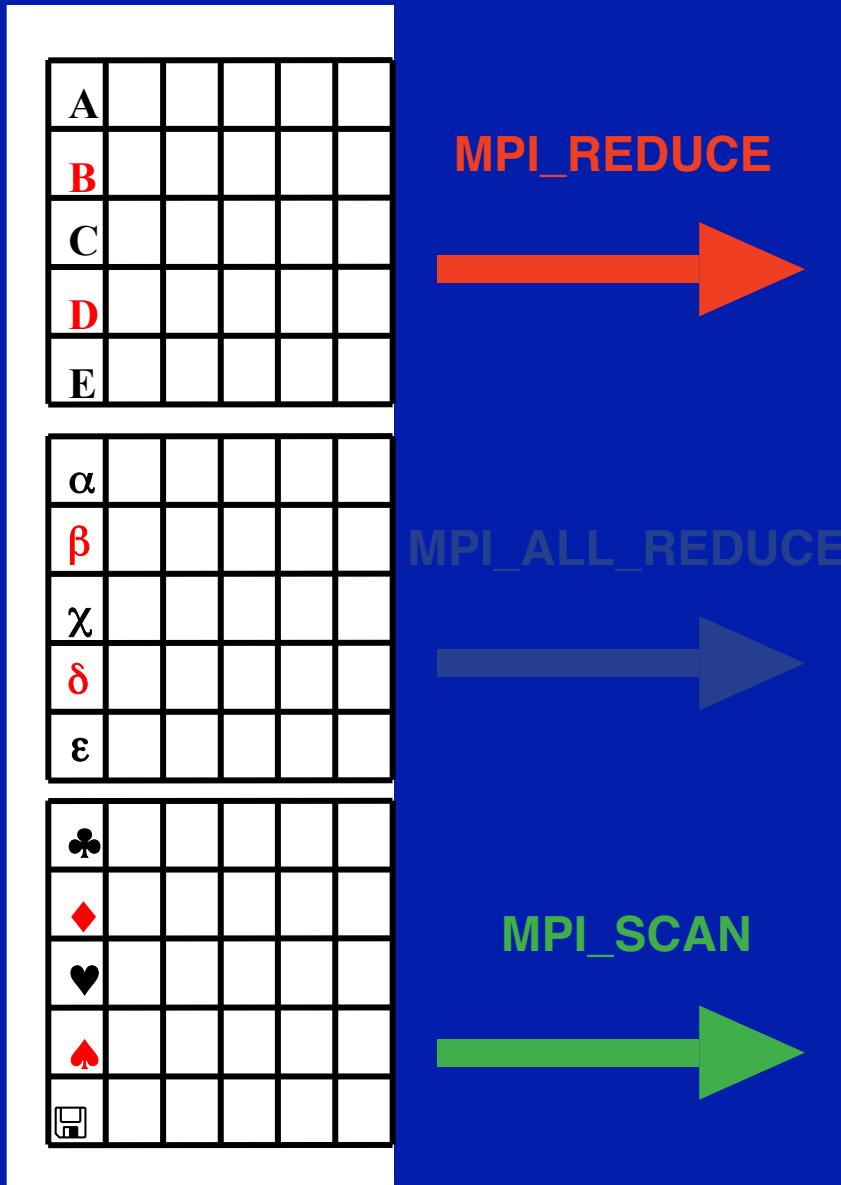
- ◆ Four Processors where each has a send buffer of size 2

- 0 1 2 3 Processors
 - (2,4) (5,7) (0,3) (6,2) Initial Send Buffers
 - MPI_BCAST with root=2
 - (0,3) (0,3) (0,3) (0,3) Resultant Buffers
 - MPI_REDUCE with action MPI_MIN and root=0
 - (0,2) (__,_) (__,_) (__,_) Resultant Buffers
 - MPI_ALLREDUCE with action MPI_MIN and root=0
 - (0,2) (0,2) (0,2) (0,2) Resultant Buffers
 - MPI_REDUCE with action MPI_SUM and root=1
 - (__,_) (13,16) (__,_) (__,_) Resultant Buffers

Collective Computation Patterns

Processors
↑

Function F()
 $F = \text{Sum}$
MAX MIN etc.



More Examples of Collective Communication/Computation

- ◆ Four Processors where each has a send buffer of size 2

- 0 1 2 3 Processors
 - (2,4) (5,7) (0,3) (6,2) Initial Send Buffers
 - MPI_SENDRECV with 0,1 and 2,3 paired
 - (5,7) (2,4) (6,2) (0,3) Resultant Buffers
 - MPI_GATHER with root=0
 - (2,4,5,7,0,3,6,2) (__,_) (__,_) (__,_) Resultant Buffers
 - Now take four Processors where only rank=0 has send buffer
 - (2,4,5,7,0,3,6,2) (__,_) (__,_) (__,_) Initial send Buffers
 - MPI_SCATTER with root=0
 - (2,4) (5,7) (0,3) (6,2) Resultant Buffers

Data Movement (1)



| | | | | | |
|---|--|--|--|--|--|
| A | | | | | |
| B | | | | | |
| C | | | | | |
| D | | | | | |
| E | | | | | |



| | | | | | |
|---|--|--|--|--|--|
| A | | | | | |
| A | | | | | |
| A | | | | | |
| A | | | | | |
| A | | | | | |

| | | | | | |
|---|--|--|--|--|--|
| A | | | | | |
| B | | | | | |
| C | | | | | |
| D | | | | | |
| E | | | | | |

MPI_SCATTER



| | | | | | |
|---|---|---|---|---|--|
| A | B | C | D | E | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

MPI_GATHER

Examples of MPI_ALLTOALL

- ◆ All to All Communication with i 'th location in j 'th processor being sent to j 'th location in i 'th processor
- ◆ Processor 0 1 2 3
- ◆ Start (a0,a1,a2,a3) (b0,b1,b2,b3) (c0,c1,c2,c3) (d0,d1,d2,d3)
- ◆ After (a0,b0,c0,d0) (a1,b1,c1,d1) (a2,b2,c2,d2) (a3,b3,c3,d3)
- ◆ There are extensions MPI_ALLTOALLV to handle case where data stored in noncontiguous fashion in each processor and when each processor sends different amounts of data to other processors
- ◆ Many MPI routines have such "vector" extensions

Data Movement (2)

| | | | | | |
|---|--|--|--|--|--|
| A | | | | | |
| B | | | | | |
| C | | | | | |
| D | | | | | |
| E | | | | | |

MPI_ALL_GATHER

| | | | | | |
|---|---|---|---|---|--|
| A | B | C | D | E | |
| A | B | C | D | E | |
| A | B | C | D | E | |
| A | B | C | D | E | |
| A | B | C | D | E | |

| | | | | | |
|---|---|---|---------------|---|--|
| A | a | 1 | α | ♣ | |
| B | b | 2 | β | ♦ | |
| C | c | 3 | χ | ♥ | |
| D | d | 4 | δ | ♠ | |
| E | e | 5 | ε | 💾 | |

MPI_ALL_TO_ALL

| | | | | | |
|----------|---------|--------|----------|---------------|--|
| A | B | C | D | E | |
| a | b | c | d | e | |
| 1 | 2 | 3 | 4 | 5 | |
| α | β | χ | δ | ε | |
| ♣ | ♦ | ♥ | ♠ | 💾 | |

List of Collective Routines

Allgather
Alltoall
Bcast
Reduce
Scatter

Allgatherv
Alltoally
Gather
ReduceScatter
Scatterv

Allreduce
Barrier
Gatherv
Scan

- ◆ “ALL” routines deliver results to all participating processes
- ◆ Routines ending in “V” allow different sized inputs on different processors

Example Fortran: Performing a Sum

```
call MPI_COMM_RANK( comm, rank, ierr )
if (rank .eq. 0) then
    read *, n
end if
call MPI_BCAST(n, 1, MPI_INTEGER, 0, comm, ierr )
# Each process computes its range of numbers to sum
lo = rank*n+1
hi = lo+n-1
sum = 0.0d0
do i = lo, hi
    sum = sum + 1.0d0 / i
end do
call MPI_REDUCEALL( sum, sumout, 1, MPI_DOUBLE,
&                      MPI_ADD_DOUBLE, comm, ierr)
```

Example C: Computing Pi

```
◆ #include "mpi.h"
◆ #include <math.h>
◆ int main (argc, argv)
◆ int argc; char *argv[];
◆ {
◆   int n, myid, numprocs, i, rc;
◆   double PI25DT = 3.14159265358979323842643;
◆   double mypi, pi, h, sum, x, a;
◆
◆   MPI_Init(&argc, &argv);
◆   MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
◆   MPI_Comm_rank (MPI_COMM_WORLD, &myid);
```

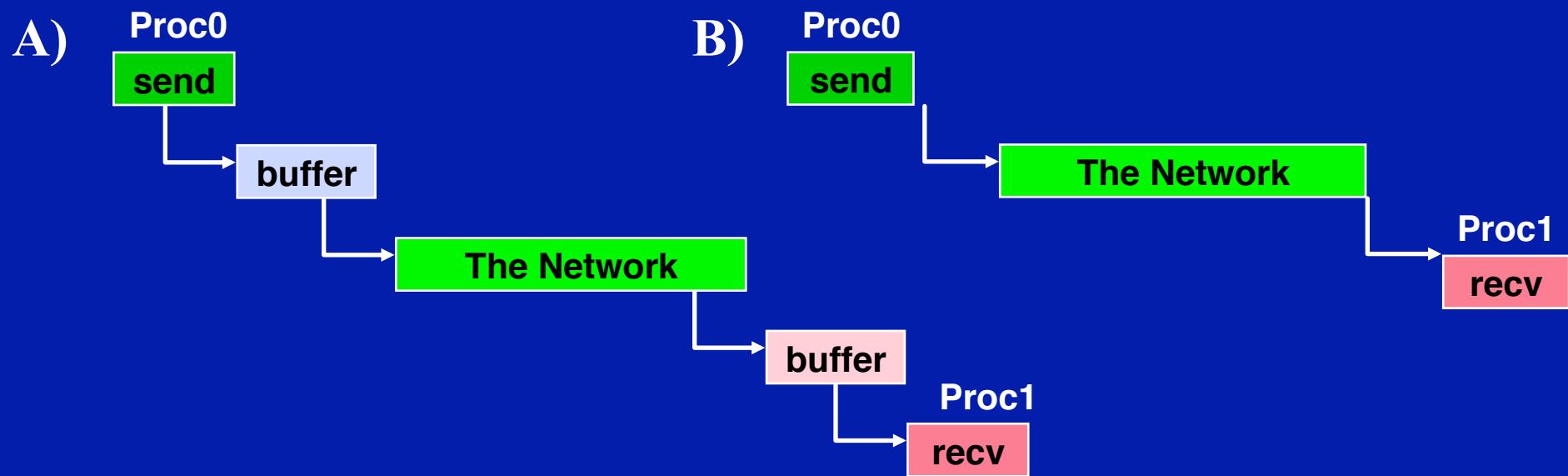
Pi Example continued

- ◆ { if (myid == 0)
 - ◆ { printf ("Enter the number of intervals: (0 quits) ");
 - ◆ scanf ("%d", &n); }
- ◆ MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMMWORLD);
- ◆ if (n == 0) break;
- ◆ h = 1.0 / (double) n;
- ◆ sum = 0.0;
- ◆ for (i = myid+1; i <= n; i += numprocs)
 - ◆ { x = h * ((double) i - 0.5); sum += 4.0 / 1.0 + x*x); }
- ◆ mypi = h * sum;
- ◆ MPI_Reduce (&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMMWORLD);
- ◆ if (myid == 0)
 - ◆ printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI35DT)); }
- ◆ MPI_Finalize; }

The many Sending and Receiving Functions

Buffering Issues

- ◆ Where does data go when you send it?
 - Multiple buffer copies, as in A)?
 - Straight to the network, as in B)?
- ◆ B) is more efficient than A), but not always safe as send waits for receive in blocking mode



Avoiding Buffering Costs

- ◆ Copies are not needed if
 - Send does not return until the data is delivered, or
 - The data is not touched after the send
- ◆ MPI provides modes to arrange this
 - Synchronous: Do not return until recv is posted
 - Ready: Matching recv is posted before send
 - Buffered: If you really want buffering
- ◆ When using asynchronous communication send functions, use MPI_Wait or MPI_WaitAll before reusing the buffer to ensure that all data has been safely transferred on its way.

Combining Blocking and Send Modes

- ◆ All combinations are legal
 - Red are fastest, Blue are slow

| | Blocking | Nonblocking |
|-------------|------------------|-------------------|
| Normal | MPI_SEND | MPI_ISEND |
| Buffering | MPI_BSEND | MPI_IBSEND |
| Ready | MPI_RSEND | MPI_IRSEND |
| Synchronous | MPI_SSEND | MPI_ISSEND |

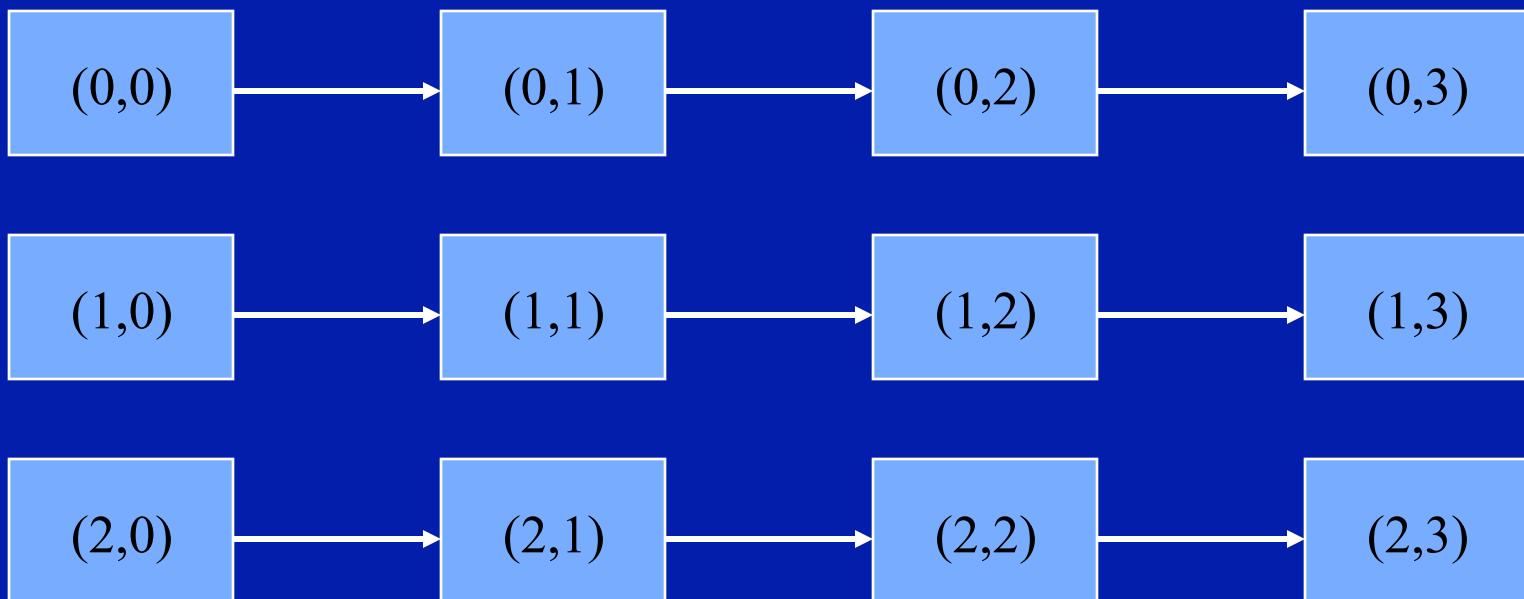
More on Send/Receive

- ◆ Note modes are only for SEND – not receive which **only has blocking MPI_RECV and non blocking MPI_IRecv**
- ◆ Note that the sending and receiving of messages synchronizes the different nodes of a parallel computer
 - So waiting in an MPI_RECV for the sent data causes the sending and receiving nodes to synchronize
 - The full pattern of sending and receiving then propagates this synchronization to ensure all nodes are indeed synchronized
- ◆ It is easy to write inconsistent code
 - In A: SEND to B: RECV from A;
 - In B: SEND to A; RECV from B will hang in some MPI implementations (Interchange SEND and RECV in one node to make it safe or better use **MPI_SENDRECV**)

TOPOLOGIES

Cartesian Topologies

- ◆ MPI provides routines to provide structure to collections of processes. Although it also has graph topologies, here we concentrate on cartesian.
- ◆ A Cartesian topology is a mesh
- ◆ Example of a 3×4 mesh with arrows pointing at the right neighbors:



Defining a Cartesian Topology

- ◆ The routine **MPI_Cart_create** creates a Cartesian decomposition of the processes, with the number of dimensions given by the **ndim** argument. It returns a new **communicator** (in **comm2d** in example below) with the same processes as in the input **communicator**, but different topology.
- ◆ **ndim** = 2;
- ◆ **dims[0]** = 3; **dims[1]** = 4;
- ◆ **periods[0]** = 0; **periods[1]** = 0; // periodic is false
- ◆ **reorder** = 1; // reordering is true
- ◆ **ierr** = **MPI_Cart_create** (**MPI_COMM_WORLD**, **ndim**,
dims, **periods**, **reorder**, **&comm2d**);
 - where **reorder** specifies that it's o.k. to reorder the default process rank in order to achieve a good embedding (with good communication times between

MPI_Cart_coords and MPI_Cart_rank

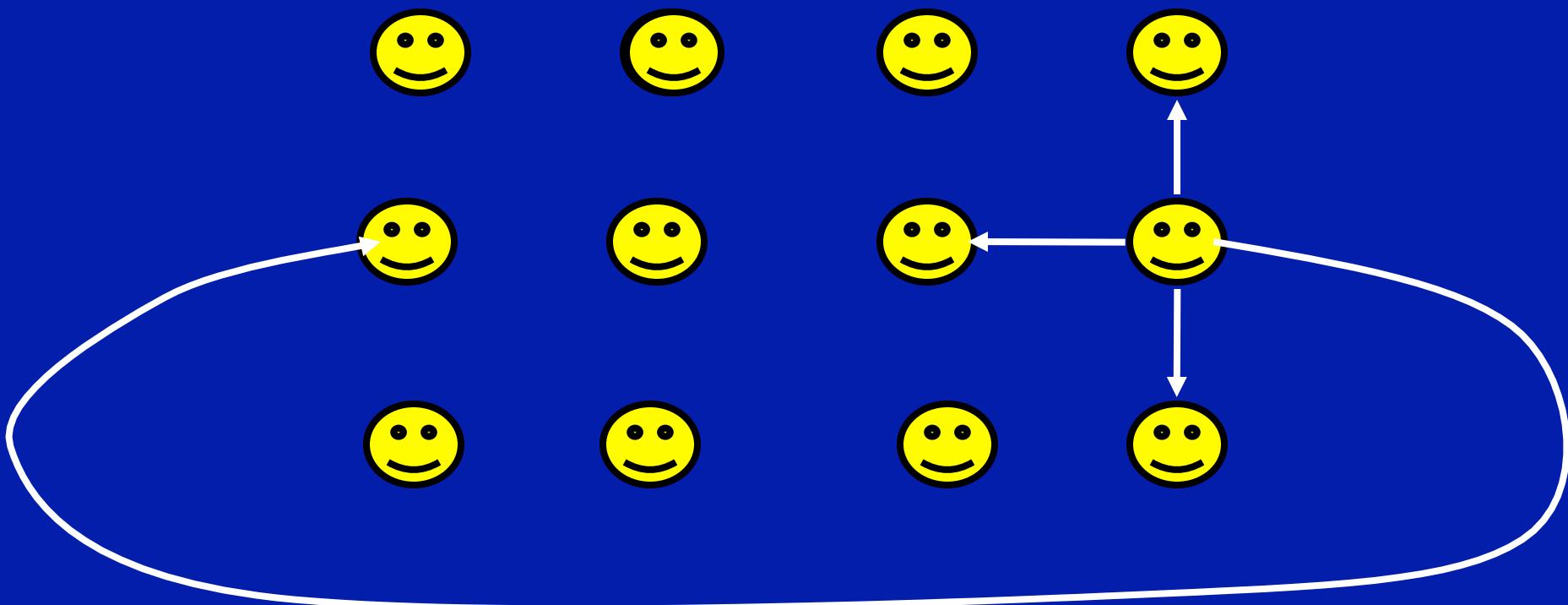
- ◆ Given the **rank** of the process in **MPI_COMM_WORLD**, this routine gives a two element (for two dimensional topology) array (**coords** in example below) with the **(i, j)** coordinates of this process in the new cartesian communicator.
 - ierr = **MPI_Cart_coords** (**comm2d, rank, ndim, coords**);
 - **coords[0]** and **coords[1]** will be the **i** and **j** coordinates.
- ◆ Given the **coords** of a process, this routine gives the rank number in the communicator.
 - ierr = **MPI_Cart_rank** (**comm2d, coords, &rank**);

Who are my neighbors?

- ◆ The routine **MPI_Cart_shift** finds the neighbors in each direction of the new communicator.
- ◆ **dir = 0;** // in C 0 for columns, 1 for rows
- ◆ // in Fortran, it's 1 and 2
- ◆ **disp = 1;** // specifies first neighbor to the right and left
- ◆ **ierr = MPI_Cart_shift (comm2d, dir, disp, &nbrbottom,**
&nbrtop);
- ◆ This returns the process numbers (ranks) for a communication of the bottom and top neighbors.
- ◆ Typically, the neighbors are used with send/recv to exchange data.
- ◆ If a process in a non-periodic mesh is on the border and has no neighbor, then the value **MPI_PROCNULL** is returned. This process value can be used in a send/recv, but it will have no effect.

Periodic meshes

- In a periodic mesh, as shown below the processes at the edge of the mesh wrap around in their dimension to find their neighbors. The right neighbor is wrapped



Communication in Sub-Grids

- ◆ Suppose that you have an algorithm, such as matrix multiply, that requires you to communicate within one row or column of a 2D grid.
 - For example, broadcast a value to all processes in one row.
- ◆ **MPI_Comm rowcomm;**
freecoords[0] = 0; freecoords[1] = 1;
ierr = MPI_Cart_sub(comm2d, freecoords, &rowcomm)
- ◆ Defines nrow new communicators, each with the processes of that row.
 - The array freecoords has boolean values specifying whether the elements of that dimension “belong” to the communicator; in example “1th” dimension is in returned sub-grid (note sub-grid can have >1 included dimension)
 - You return sub-grid CONTAINING processor doing call
- ◆ if bcastroot is defined as the root processor in each row, broadcast a value along rows.

Message Buffers and Datatypes

What Gets Sent: The Buffer

- ◆ First generation message passing systems only allowed one to transmit information originating in a **contiguous array of bytes**
 - Hid the real data structure from hardware and programmer
 - » Might make it **hard** to provide **efficient implementations** as implied a lot of expensive memory accesses
 - Required pre-packing dispersed data, e.g.:
 - » **Rows** (in **Fortran**, columns in C) of a matrix must be **transposed before transmission**
 - Prevented convenient communication between machines with different data representations

Generalizing the Buffer in MPI

- ◆ MPI specifies the buffer by *starting address*, *datatype*, and *count*
 - starting address is obvious
 - datatypes are constructed recursively from
 - » Elementary (all C and Fortran datatypes)
 - » Contiguous array of datatypes
 - » Strided blocks of datatypes
 - » Indexed array of blocks of datatypes
 - » General structures
 - count is number of datatype elements

Advantages of Datatypes

- ◆ Combinations of **elementary datatypes** into a derived user defined **datatype** allows clean communication of collections of disparate types in a single MPI call.
- ◆ Elimination of **length** (in bytes) in favor of **count** (of items of a given type) is clearer
- ◆ Specifying **application-oriented layouts** allows maximal use of special hardware and optimized memory use
- ◆ However this wonderful technology is problematical in **Java** where layout of data structures in memory is not defined in most cases
 - Java's **serialization** subsumes user defined datatypes as a general way of packing a class of disparate types into a message that can be sent between heterogeneous computers

Motivation for Derived Datatypes in MPI

- ◆ These are an elegant solution to a problem we struggled with a lot in the early days -- all message passing is naturally built on buffers holding contiguous data
- ◆ However often (usually) the data is not stored contiguously. One can address this with a set of small MPI_SEND commands but we want messages to be as big as possible as latency is so high
- ◆ One can copy all the data elements into a single buffer and transmit this but this is tedious for the user and not very efficient
 - It has extra memory to memory copies which are often quite slow
- ◆ So derived datatypes can be used to set up arbitrary memory templates with variable offsets and primitive datatypes. Derived datatypes can then be used in "ordinary" MPI calls in place of primitive datatypes MPI_REAL MPI_FLOAT etc.

Derived Datatype Basics

- ◆ Derived Datatypes should be declared integer in Fortran and MPI_Datatype in C
- ◆ Generally have form { (type0,disp0), (type1,disp1) ... (type(n-1),disp(n-1)) } with list of primitive data types typei and displacements (from start of buffer) dispi
- ◆ call MPI_TYPE_CONTIGUOUS (count, oldtype, newtype, ierr)
 - creates a new datatype newtype made up of count repetitions of old datatype oldtype
- ◆ one must use call MPI_TYPE_COMMIT(derivedtype, ierr) before one can use the type derivedtype in a communication call
- ◆ call MPI_TYPE_FREE(derivedtype, ierr) frees up space used by this derived type

Simple Example of Derived Datatype

- ◆ integer **derivedtype**, ...
- ◆ call **MPI_TYPE_CONTIGUOUS(10, MPI_REAL, derivedtype, ierr)**
- ◆ call **MPI_TYPE_COMMIT(derivedtype, ierr)**
- ◆ call **MPI_SEND(data, 1, derivedtype, dest, tag, MPI_COMM_WORLD, ierr)**
- ◆ call **MPI_TYPE_FREE(derivedtype, ierr)**
- ◆ is equivalent to simpler single call
- ◆ call **MPI_SEND(data, 10, MPI_REAL, dest, tag, MPI_COMM_WORLD, ierr)**
- ◆ and each sends 10 contiguous real values at location **data** to process **dest**

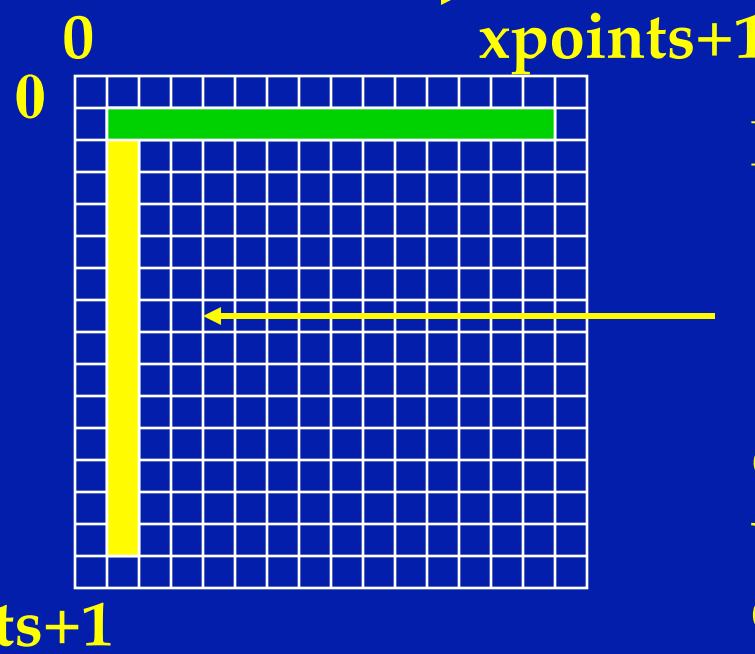
Derived Datatypes: Vectors

- ◆ **MPI_TYPE_VECTOR** (**count, blocklen, stride, oldtype, newtype, ierr**)
 - IN **count** Number of blocks to be added
 - IN **blocklen** Number of elements in block
 - IN **stride** Number of elements (NOT bytes) between start of each block
 - IN **oldtype** Datatype of each element
 - OUT **newtype** Handle(pointer) for new derived type

Example of Vector type

- ◆ Suppose in C, we have an array
 - **phi [ypoints+2] [xpoints+2]**
 - where we want to send rows and columns of elements from **1 : nxblock** and **1 : nyblock**
 - in C, arrays are stored row major order (Fortran is column major)

Contiguous elements

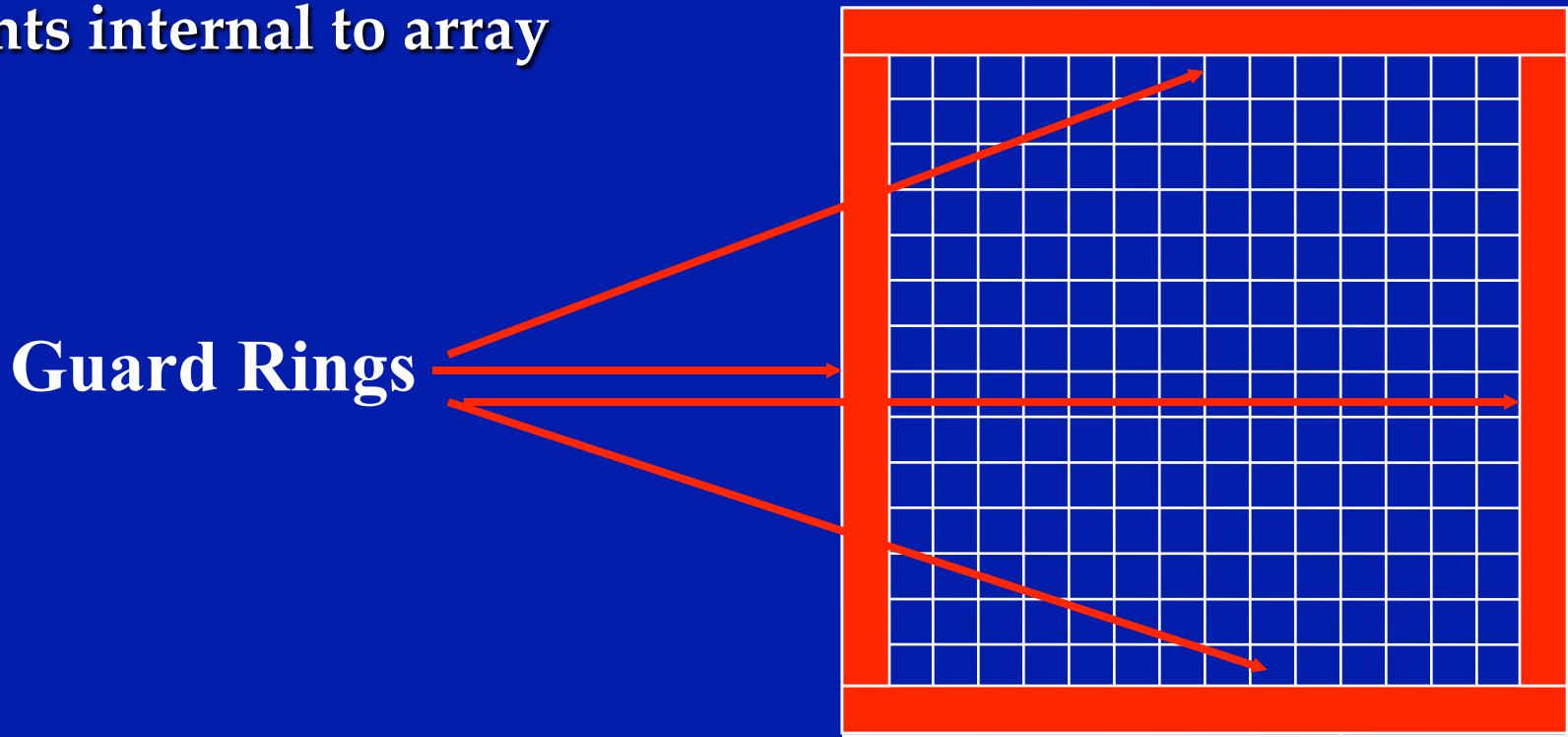


```
MPI_Type_vector  
(xpoints, 1, ypoints+2,  
 MPI_DOUBLE, &strided);
```

defines a type called strided which refers to the column of elements

Why is this interesting?

- ◆ In Jacobi like algorithms, each processor stores its own **xpoints by ypoints** array of variables as well as guard rings containing the rows and columns from neighbours. One loads these guard rings at start of computation iteration and only updates points internal to array



Derived Datatypes: Indexed

- ◆ Array of indices, useful for gather/scatter
- ◆ **MPI_TYPE_INDEXED** (**count, blocklens, indices, oldtype, newtype, ierr**)
 - IN **count** Number of blocks to be added
 - IN **blocklens** Number of elements in each block -- an array of length **count**
 - IN **indices** Displacements (an array of length **count**) for each block
 - IN **oldtype** Datatype of each element
 - OUT **newtype** Handle(pointer) for new derived type

Jacobi Iteration

an example

Designing MPI Programs

◆ Partitioning

- Before tackling MPI

◆ Communication

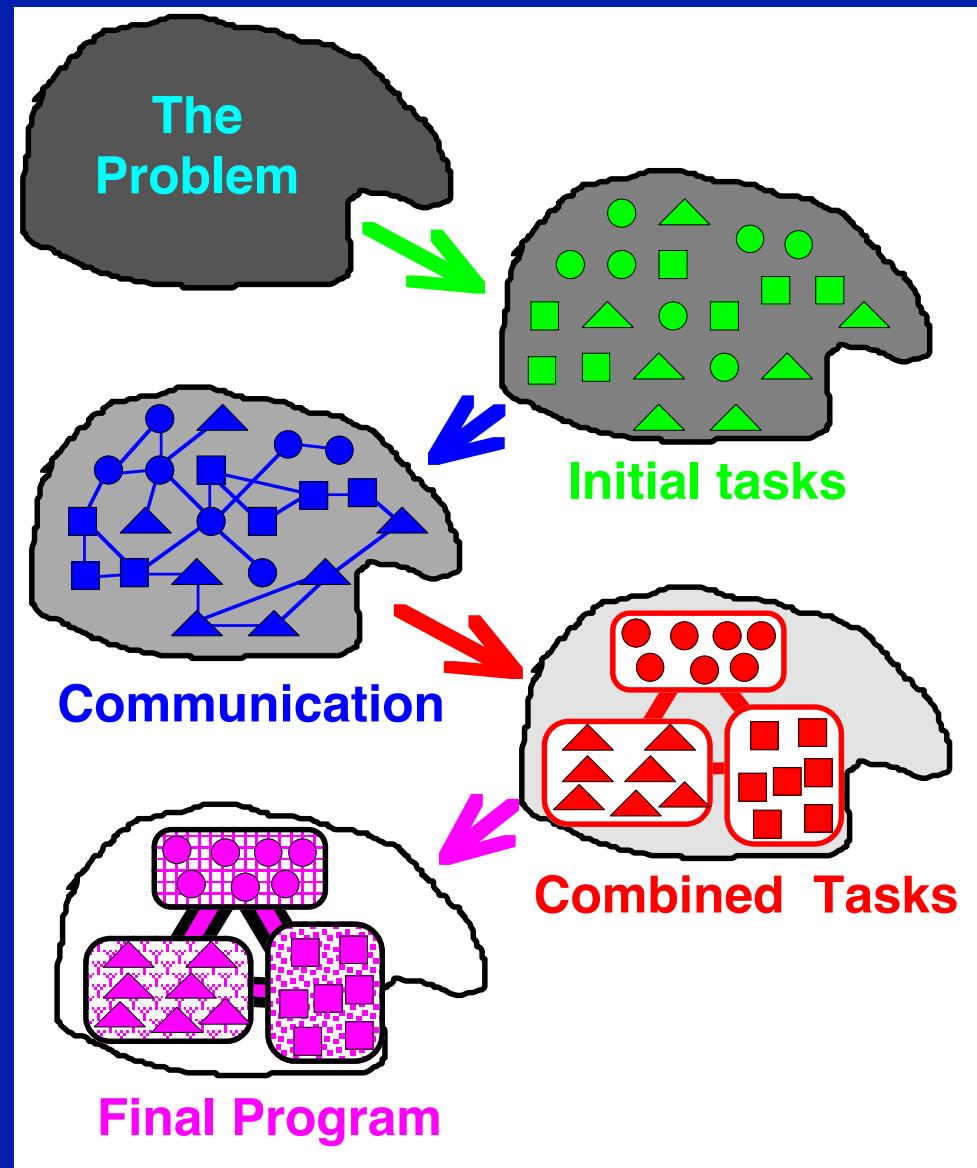
- Many point to collective operations

◆ Agglomeration

- Needed to produce MPI processes

◆ Mapping

- Handled by MPI



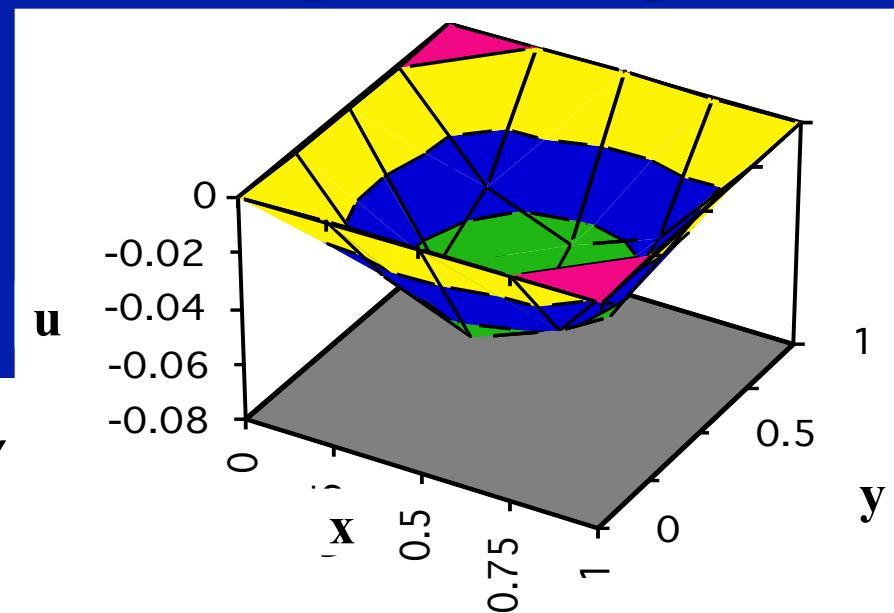
Jacobi Iteration: The Problem

- Used to numerically solve a Partial Differential Equation (PDE) on a square mesh -- below is Poisson's Equation
- Method:
 - Update each mesh point by the average of its neighbors
 - Repeat until converged

This is right hand side
 $f(x,y)$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -2x^2 + 2x - 2y^2 + 2y$$

$$u = 0 \text{ if } x = 0, x = 1, y = 0, \text{ or } y = 1$$



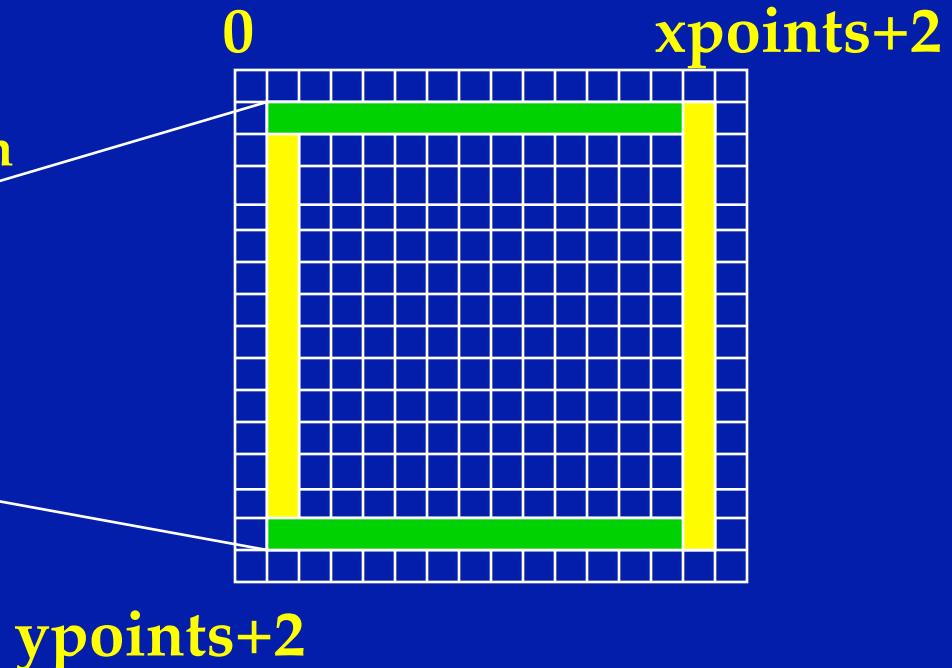
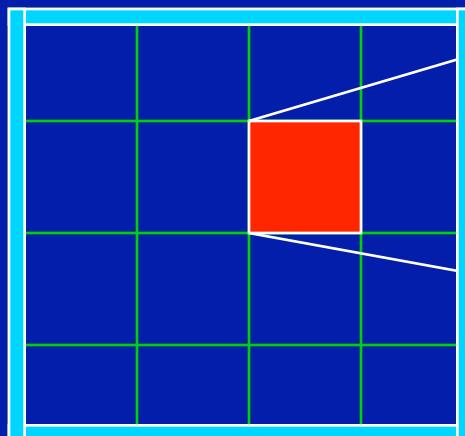
Jacobi Iteration: MPI Program Design

- ◆ **Partitioning** is simple
 - Every point is a micro-task
- ◆ **Communication** is simple
 - 4 nearest neighbors in **Cartesian mesh**
 - Reduction for convergence test
- ◆ **Agglomeration** works along dimensions
 - **1-D packing** for high-latency machines (as minimizes number of messages)
 - **2-D packing** for others (most general as minimizes information sent)
 - **One process per processor** practically required

Jacobi Iteration: MPI Program Design

- ◆ Mapping: Cartesian grid supported by MPI virtual topologies
- ◆ For generality, write as the 2-D version
 - Create a $1 \times P$ (or $P \times 1$) grid for 1-D version
- ◆ Adjust array bounds, iterate over local array
 - For convenience, include shadow region to hold communicated values (not iterated over)

nx by ny points in a
 npx by npy decomposition,
boundary values define problem



Jacobi Iteration: C MPI Program Sketch

```
/* sizes of data and data files */  
int NDIM = 2;  
int xpoints = nx/npx; int ypoints = ny/npy;  
double phi[ypoints+2][xpoints+2], oldphi[ypoints  
+2][xpoints+2];  
  
/* communication variables */  
int rank; int rankx, ranky;  
int coords[NDIM];  
int reorder = 0;  
int dims[NDIM], periods[NDIM];  
MPI_Comm comm2d;  
MPI_Datatype contig, strided;
```

Jacobi Iteration: create topology

```
MPI_Comm_rank( MPI_COMM_WORLD, &rank );
MPI_Comm_size( MPI_COMM_WORLD, &size );
```

```
periods[0] = 0;  periods[1] = 0;
dims[0] = npy;  dims[1] = npx;
```

```
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods,
                 reorder, &comm2d);
```

```
MPI_Cart_coords(comm2d, rank, 2, coords);
ranky = coords[0];
rankx = coords[1];
```

```
MPI_Cart_shift(comm2d, 0, 1, &bottomneighbor, &topneighbor);
MPI_Cart_shift(comm2d, 1, 1, &leftneighbor, &rightneighbor);
```

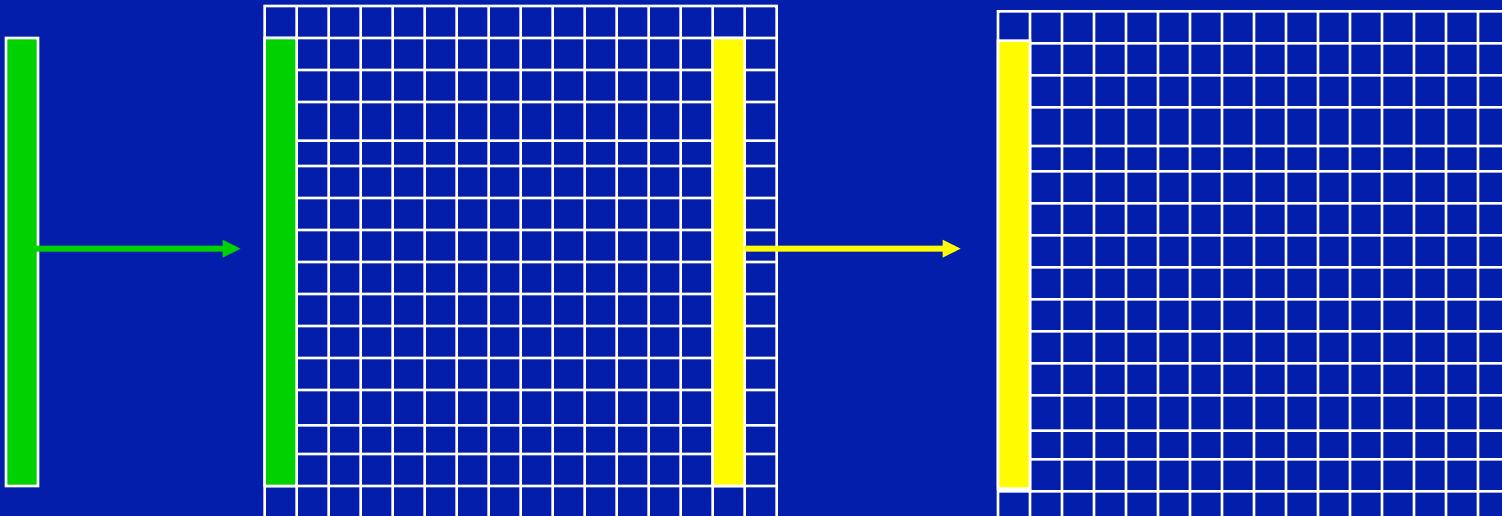
Jacobi iteration: data structures

```
/* message types */
MPI_Type_contiguous (ypoints, MPI_DOUBLE, &contig);
MPI_Type_vector (xpoints, 1, ypoints+2, MPI_DOUBLE, &strided);
MPI_Type_commit (&contig);
MPI_Type_commit (&strided);

/* define mask array to be true on boundary and false elsewhere in
   each processor.
   Define boundary values in phi array in each processor.
*/
```

Jacobi Iteration: send guard values

```
while (err > tol) { /* copy phi array to oldphi */  
/* communicate edge rows and columns to neighboring  
processor to put in their guard rings */  
/* Send right boundary to right neighbor  
and receive left ghost vector in return from left neighbor */  
MPI_Sendrecv (&(oldphi[1][xpoints]), 1, strided,  
rightneighbor, 31, &(oldphi[1][0]), 1, strided,  
leftneighbor, 31, comm2d, &status);
```



Remaining communication

```
/* Send left boundary in each processor to left neighbor */
MPI_Sendrecv (&(oldphi[1][1]), 1, strided, leftneighbor, 30,
              &(oldphi[1][xpoints+1]), 1, strided, rightneighbor, 30,
              comm2d, &status);

/* Send top boundary to top neighbor */
MPI_Sendrecv (&(oldphi[1][1]), 1, contig, topneighbor, 40,
              &(oldphi[ypoints+1][1]), 1, contig, bottomneighbor, 40,
              comm2d, &status);

/* Send bottom boundary to bottom neighbor */
MPI_Sendrecv (&(oldphi[ypoints][1]), 1, contig,
              bottomneighbor, 41, &(oldphi[0][1]), 1, contig,
              topneighbor, 41, comm2d, &status);
```

Jacobi Iteration: update and error

```
for (j = 1; j <= xpoints; j++)  
{  
    for (i = 1; i <= ypoints; i++)  
    {  
        if (mask[i][j]) {  
            phi[i][j] = 0.25 * (oldphi[i-1][j] +  
                oldphi[i+1][j]  
                + oldphi[i][j-1] +  
                oldphi[i][j+1]);  
            diff = max(diff, abs(phi[i][j] - oldphi[i][j]));  
        }  
    }  
}  
/* maximum difference over all processors */  
MPI_Allreduce(&diff, &err, 1, MPI_DOUBLE, MPI_MAX,  
comm2d);  
if (err < ((double)TOLERANCE)) done = 1;  
}
```

The MPI Timer

- ◆ The elapsed (wall-clock) time between two points in an MPI program can be computed using **MPI_Wtime**:
 - **double t1, t2;**
 - **t1 = MPI_Wtime();**
 - **...**
 - **t2 = MPI_Wtime();**
 - **printf (“Elapsed time is %f \n”, t2-t1);**
- ◆ The times are local; the attribute **MPI_WTIME_IS_GLOBAL** may be used to determine if the times are also synchronized with each other for all processes in **MPI_COMM_WORLD**.

MPI-2

- ◆ The MPI Forum produced a new standard which include MPI 1.2 clarifications and corrections to MPI 1.1
- ◆ MPI-2 new topics are:
 - process creation and management, including client/server routines
 - one-sided communications (put/get, active messages)
 - extended collective operations
 - external interfaces
 - I/O
- ◆ additional language bindings for C++ and Fortran-90

I/O included in MPI-2

- ◆ Goal is to provide model for portable file system allowing for optimization of parallel I/O
 - portable I/O interface **POSIX** judged not possible to allow enough optimization
- ◆ Parallel I/O system provides high-level interface supporting transfers of global data structures between process memories and files.
- ◆ Significant optimizations required include:
 - grouping, collective buffering, and disk-directed I/O
- ◆ Other optimizations also achieved by
 - asynchronous I/O, strided accesses and control over physical file layout on disks.
- ◆ I/O access modes defined by data partitioning expressed with derived datatypes

Comments on Parallel Input/Output - I

- ◆ Parallel I/O has technical issues -- how best to optimize access to a file whose contents may be stored on N different disks which can deliver data in parallel and
- ◆ Semantic issues -- what does printf in C (and PRINT in Fortran) mean?
- ◆ The meaning of printf/PRINT is both undefined and changing
 - In my old Caltech days, printf on a node of a parallel machine was a modification of UNIX which automatically transferred data from nodes to "host e.g. node 0" and produced a single stream
 - In those days, full UNIX did not run on every node of machine
 - We introduced new UNIX I/O modes (singular and multiple) to define meaning of parallel I/O and I thought this was a great idea but it didn't catch on!!

Comments on Parallel Input/Output - II

- ◆ Today, memory costs have declined and ALL mainstream MIMD distributed memory machines whether clusters of workstations/PC's or integrated systems such as T3D/Paragon/ SP-2 have enough memory on each node to run UNIX or Windows NT
- ◆ Thus **printf** today means typically that the node on which it runs will stick it out on "standard output" file for that node
 - However this is implementation dependent
- ◆ If on other hand you want a stream of output with information in order
 - » Starting with that from **node 0**, then **node 1**, then **node 2** etc.
 - » This was default on old Caltech machines but
 - Then in general you need to communicate information from nodes 1 to N-1 to node 0 and let node 0 sort it and output in required order
- ◆ **MPI-IO** standard links I/O to **MPI** in a standard fashion