Introduction to MPI Programming

Erwin Laure

Director Max Planck Computing and Data Facility
& Technical University Munich

What does MPI stand for?

Message **P**assing **I**nterface

Why message passing?

- OpenMP does not know the concept of message passing …
- Distributed memory architectures don’t offer shared memory/address space

Contents

- Fundamentals of Distributed Memory Computing
  - Programming models
  - Issues and techniques
- MPI Concepts
- Basic MPI Programming
  - MPI program structure
  - Point-to-point communication
  - Collective operations
- Intermediate MPI
  - Datatypes
  - Communicators
  - Improving performance

Material

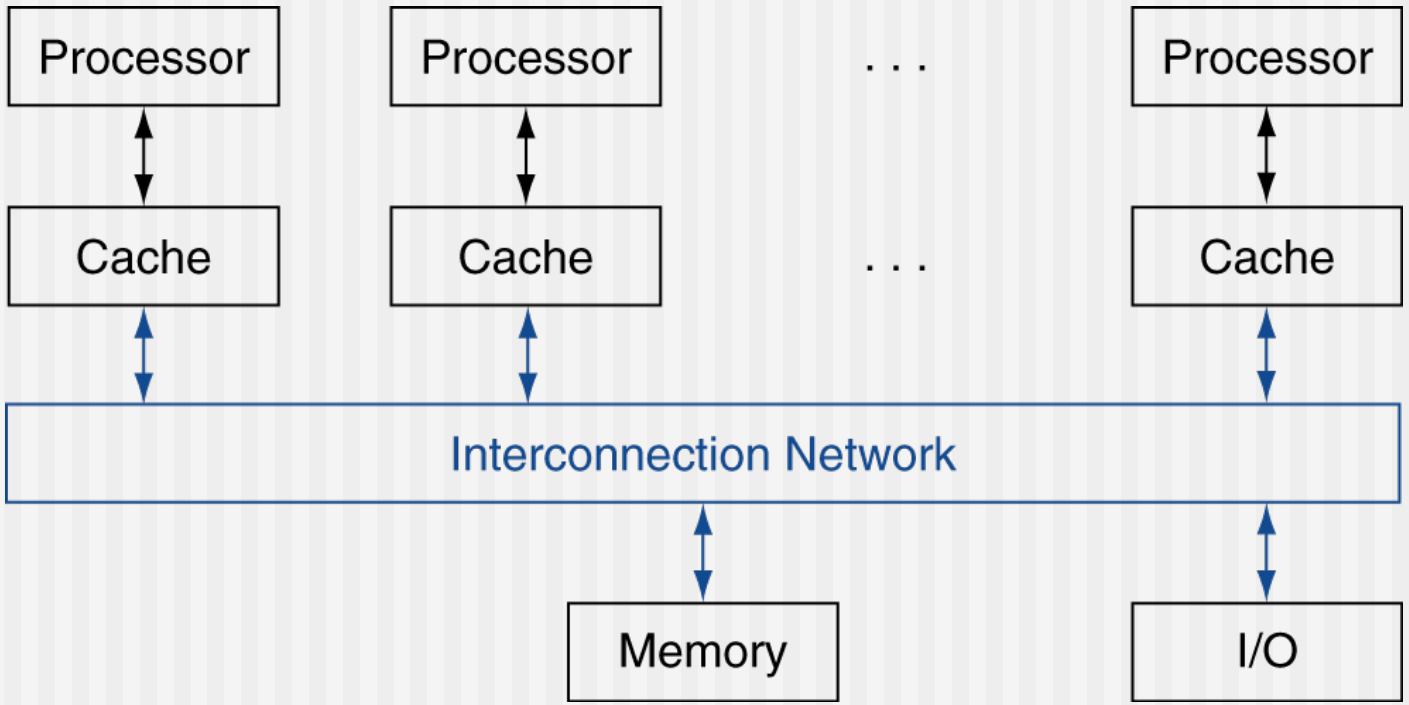
- This course is mainly based on
- Using MPI – Portable Parallel Programming with the Message-Passing Interface, W. Gropp, E. Lusk and A. Skjellum, MIT Press, 1994
- Several online tutorials:
  - http://www.mcs.anl.gov/research/projects/mpi/tutorial/
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.nccs.nasa.gov/tutorials/mpi1.pdf.gz
  - http://www.citutor.org/index.php
- Lecture notes by Michael Hanke, CSC, KTH

Recap: Computer Architecture

Shared Memory

Shared Memory Multiprocessor

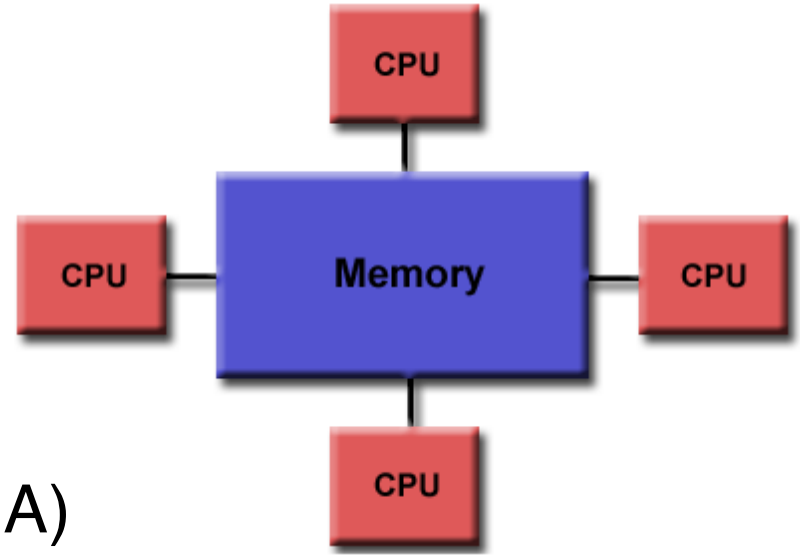
- Hardware provides single physical address space for all processors
- Global physical address space and symmetric access to all of main memory (*symmetric multiprocessor - SMP*)
- All processors and memory modules are attached to the same interconnect (bus or switched network)



Differences in Memory Access

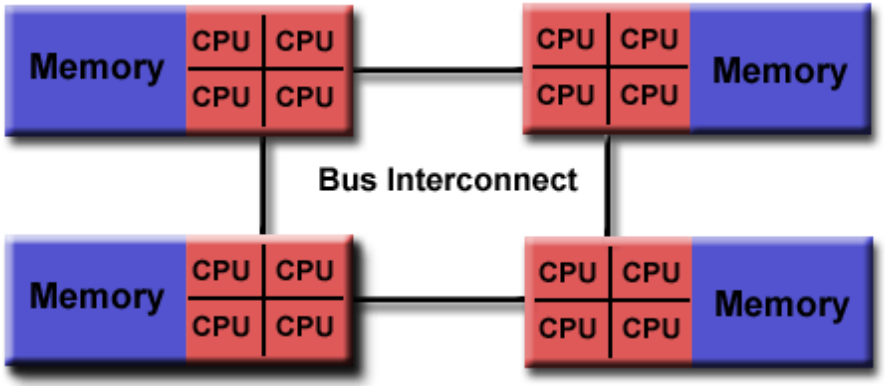
- Uniform Memory Access (UMA)

  Memory access takes about the same time independent of data location and requesting processor



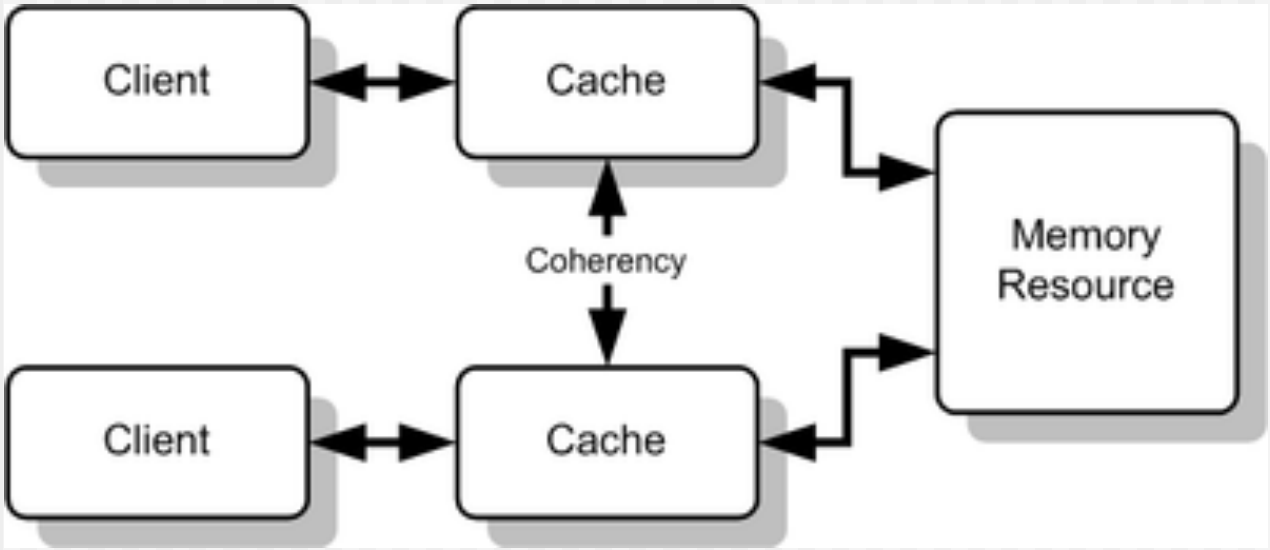
- Nonuniform memory access (NUMA)

  Memory access can differ depending on where the data is located and which processor requests the data



Cache coherence

- While main memory is shared, caches are local to individual processors
- Client B's cache might have old data since updates in client A's cache are not yet propagated
- Different cache coherency protocols to avoid this problem



Synchronization

- Access to shared data needs to be protected
  - Mutual exclusion (mutex)
  - Point-to-point events
  - Global event synchronization (barrier)

- Generic three steps:
  1. Wait for lock
  2. Acquire lock
  3. Release lock

SMP Pros and Cons

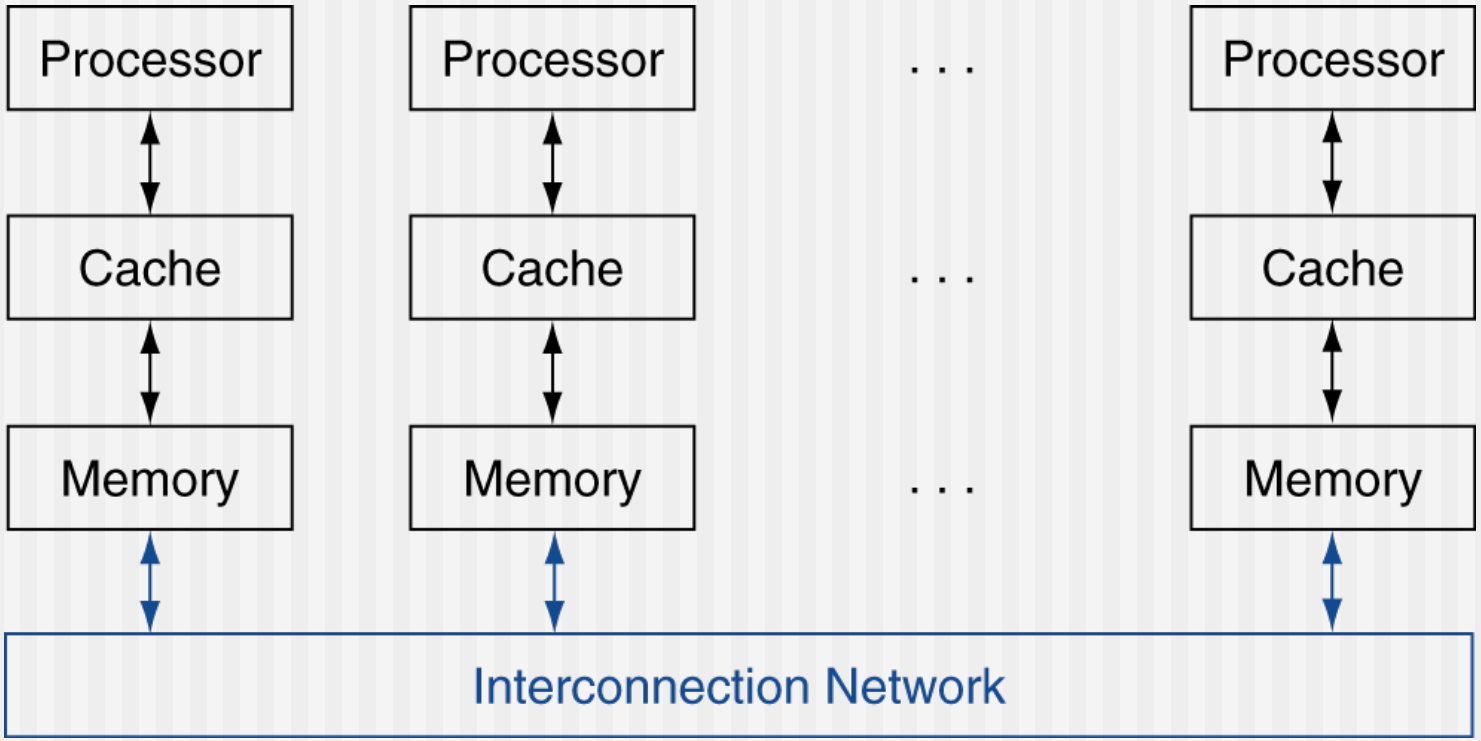
Fugaku 7,630,848 cores

- **Advantages:**
  - Global address space provides a user-friendly progra g perspective to memory
  - Data sharing between tasks is both fast and uniform du to the proximity of memory to CPUs
- **Disadvantages:**
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory Multiprocessors

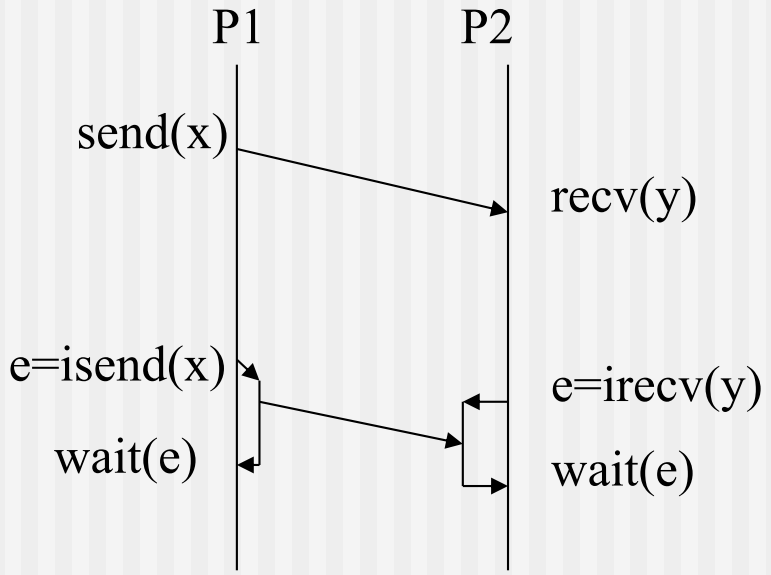
DMMPs

- Each processor has private physical address space
  - No cache coherence problem
- Hardware sends/receives messages between processors
  - Message passing



Synchronization

- Synchronization via exchange of messages
- Synchronous communication
  - Sender/receiver wait until data has been sent/received
- Asynchronous communication
  - Sender/receiver can proceed after sending/receiving has been initiated
- Higher level concepts (barriers, semaphores, …) can be constructed using send/recv primitives
  - Message passing libraries typically provide them



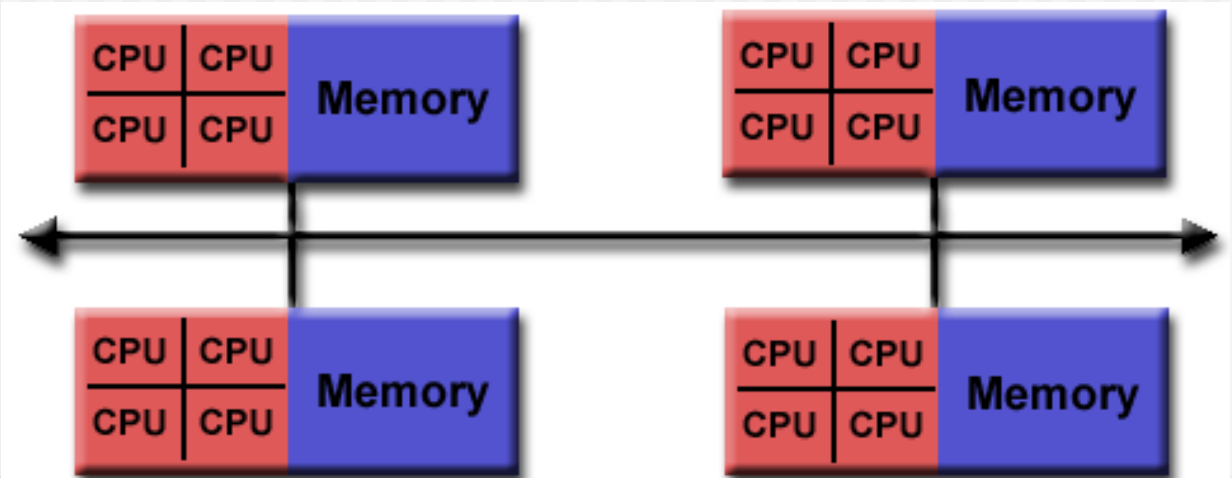
DMMPs Pros and Cons

- **Advantages:**
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- **Disadvantages:**
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Very different access times for local/non-local memory
  - Administration and software overhead (essentially N systems vs. 1 SMP)

Hybrid Approaches

Combining SMPs and DMMPs

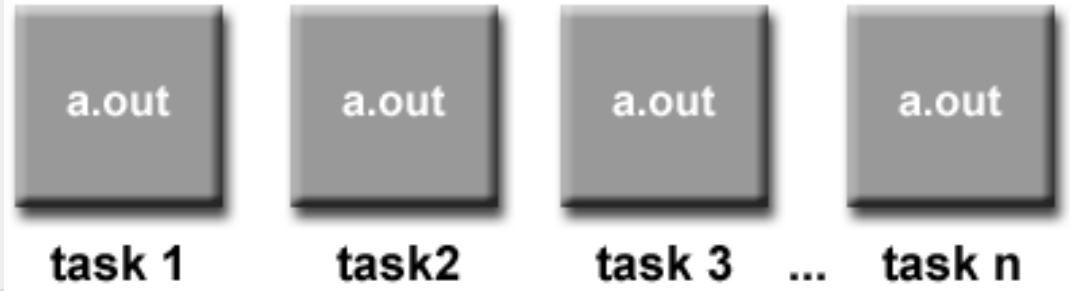
- Today, DMMPs are typically built with SMPs as building blocks
  - E.g. Dardel has two AMD CPUs with 64 cores each per DMMP node
  - Soon systems with more CPUs and many more cores will appear
    - upcoming AMD CPUS ~200 cores
- Combine advantages and disadvantages from both categories
  - Programming is more complicated due to the combination of several different memory organizations that require different treatment



Programming DMMPs

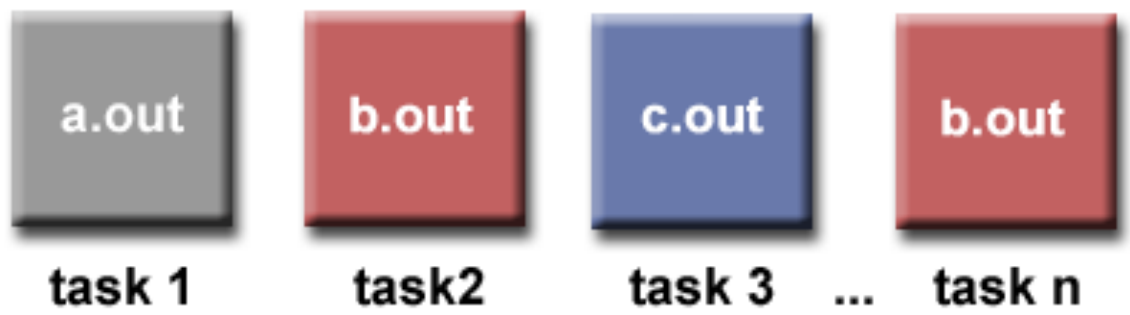
Single Program Multiple Data (SPMD)

- DMMPs are typically programmed following the SPMD model
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program. All tasks may use different data. (MIMD)
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.



Multiple Program Multiple Data (MPMD)

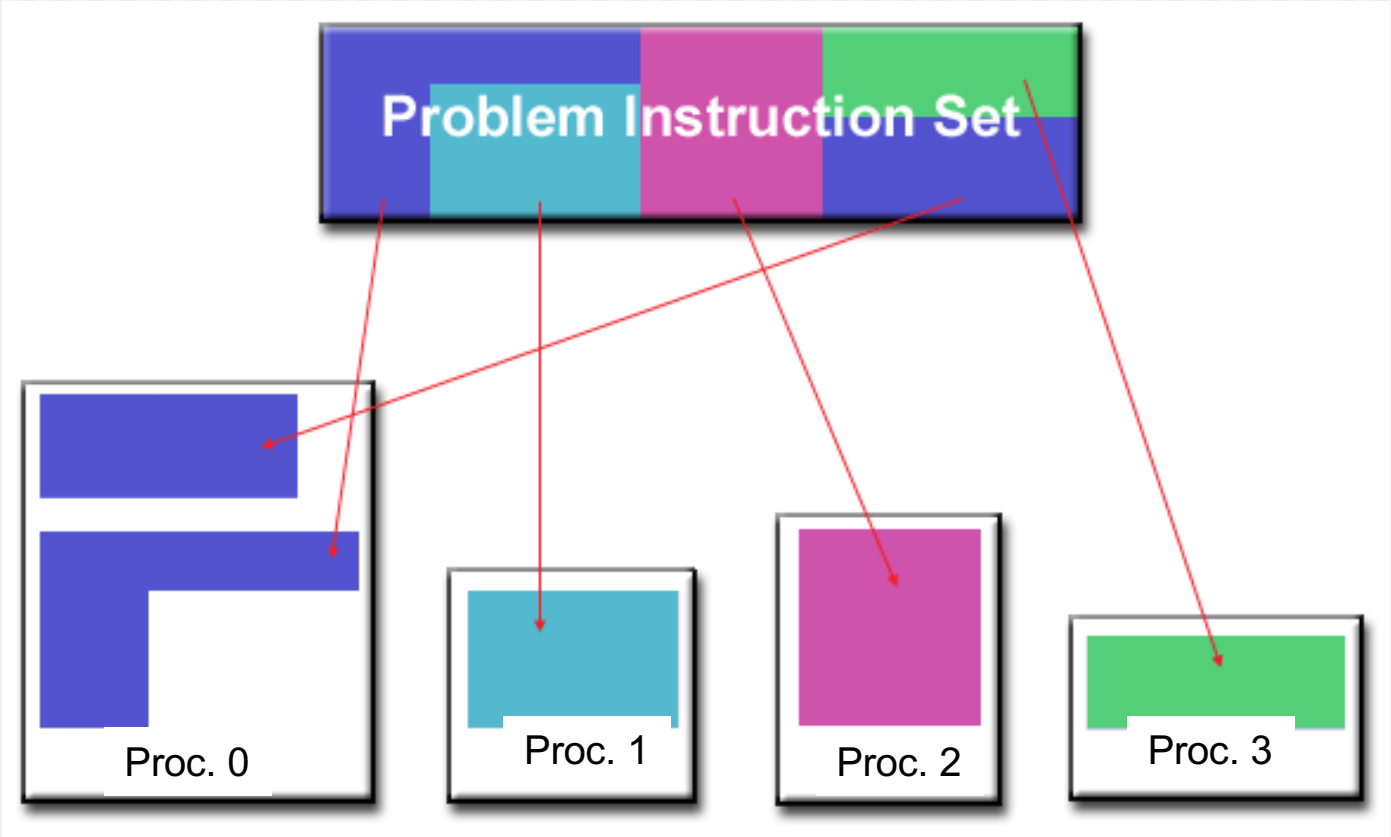
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data
- Workflow applications, multidisciplinary optimization, combination of different models



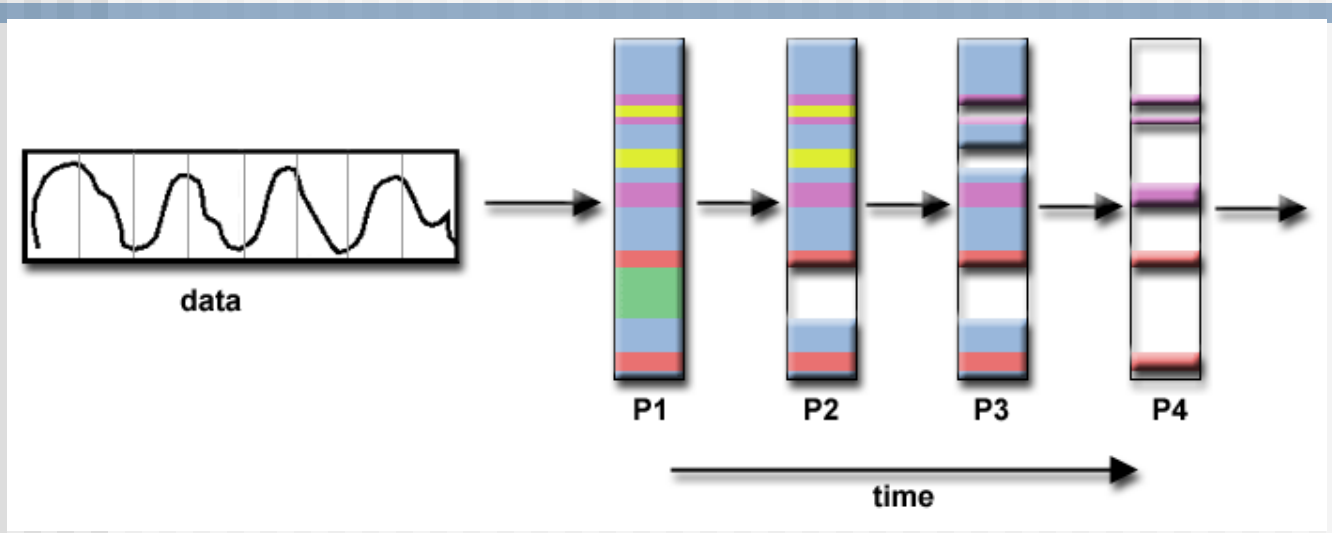
How to decompose a problem in SPMD?

Functional Decomposition

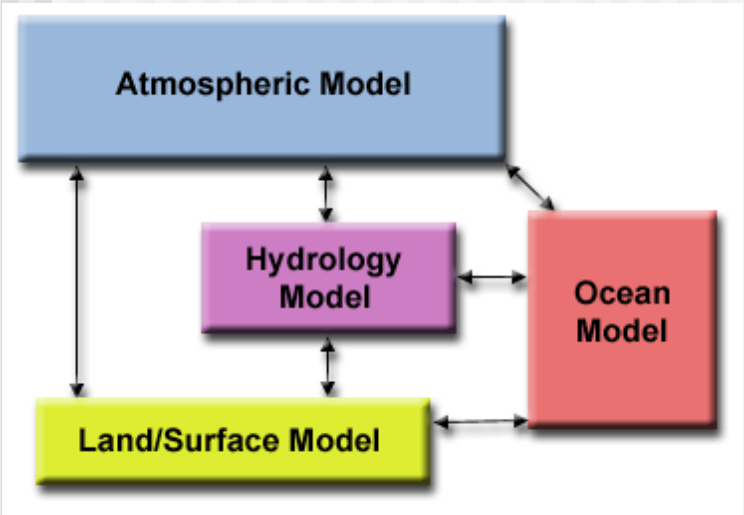
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Also called “Task Parallelism”



Task Parallelism Examples

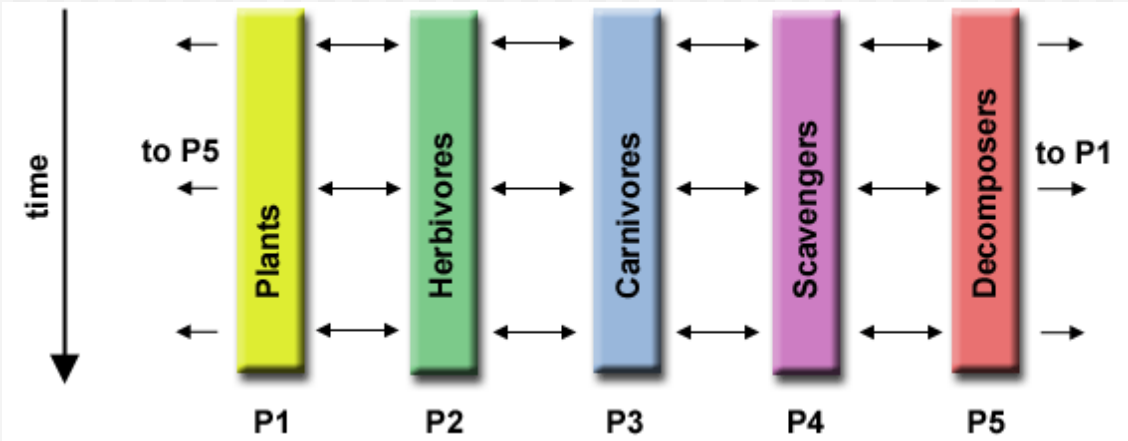


Signal filtering



Ecosystem modeling

Climate modeling

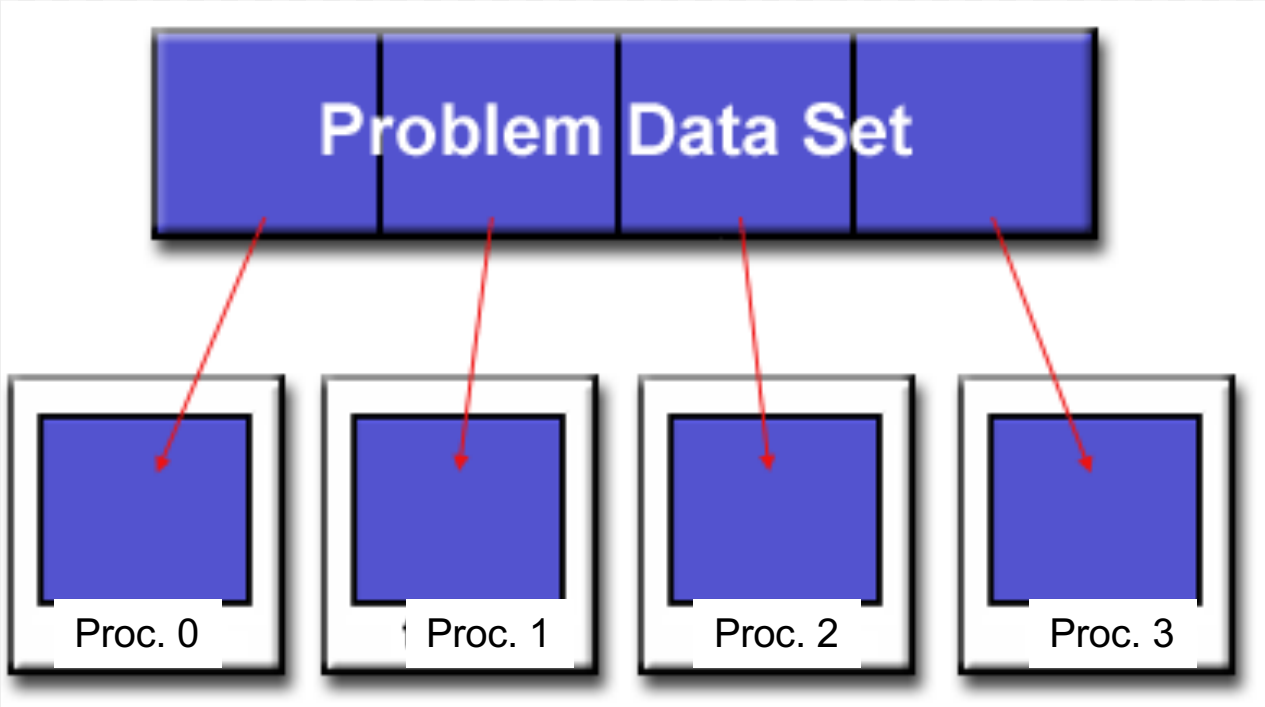


Task Parallelism Summary

- Often pipelined approaches or Master/Worker
  - Master assigns work items to its workers
- “Natural” approach to parallelism
- Typically good efficiency
  - Tasks proceed without interactions
  - Synchronization/communication needed at the end
- In practice scalability is limited
  - Problem can by split only into a finite set of different tasks

Domain Decomposition

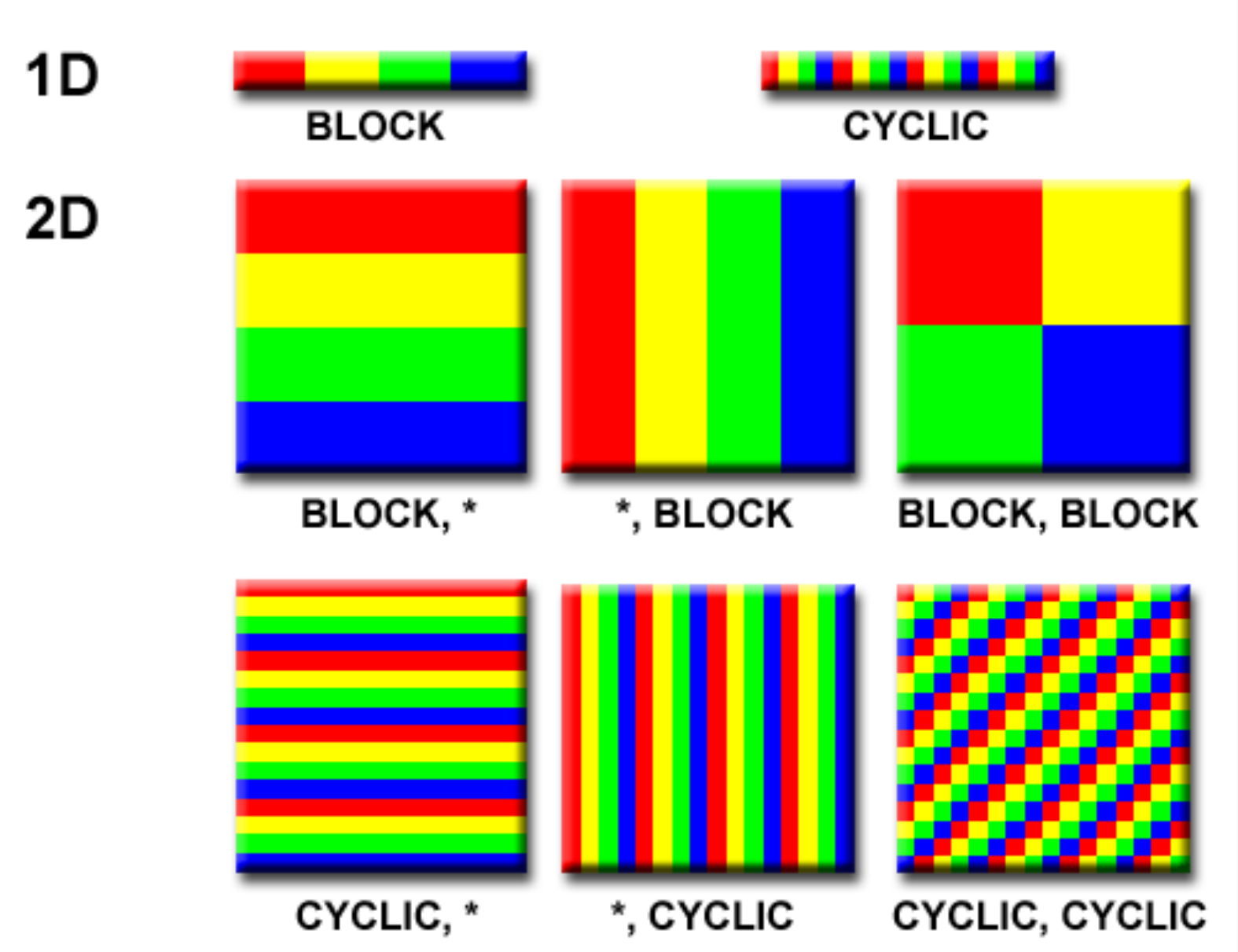
- The data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.
- Also called “Data Parallelism”



How to Partition Data

- Distribution Function:
  - *f(N)->P*; *N* denotes the data index and *P* the target processor
- Typical strategies are
  - Block
    - Distribute data in equal blocks over available processors
  - Cyclic
    - Distribute individual data items in round robin fashion over available processors
  - "*"
    - Replicate along a dimension
  - Irregular
    - Distribute data in over the processors using any kind of distribution function

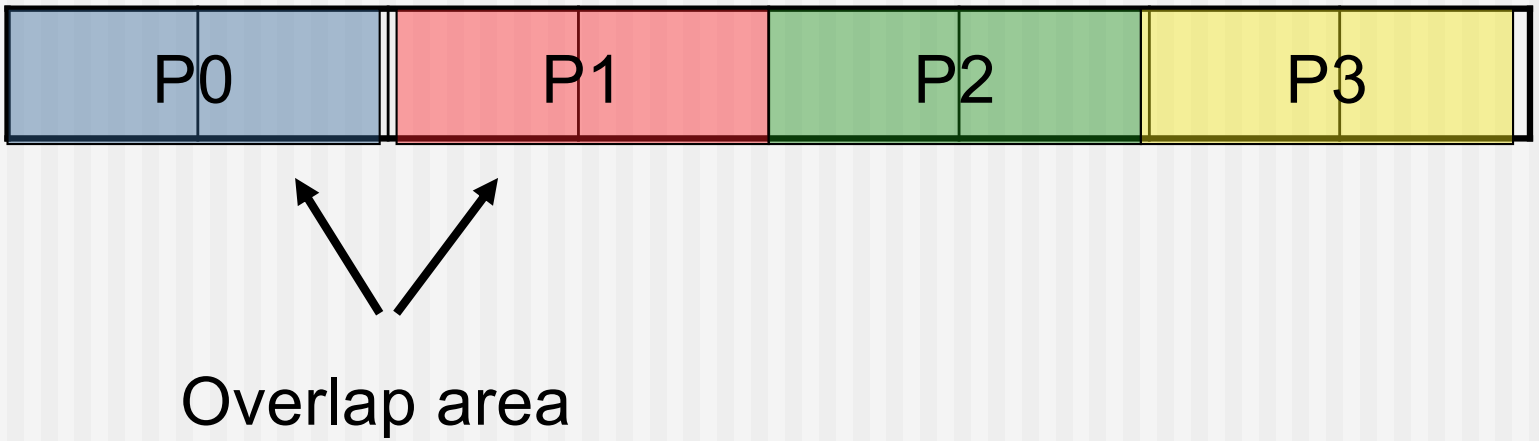
Typical Data Distributions



Access Patterns

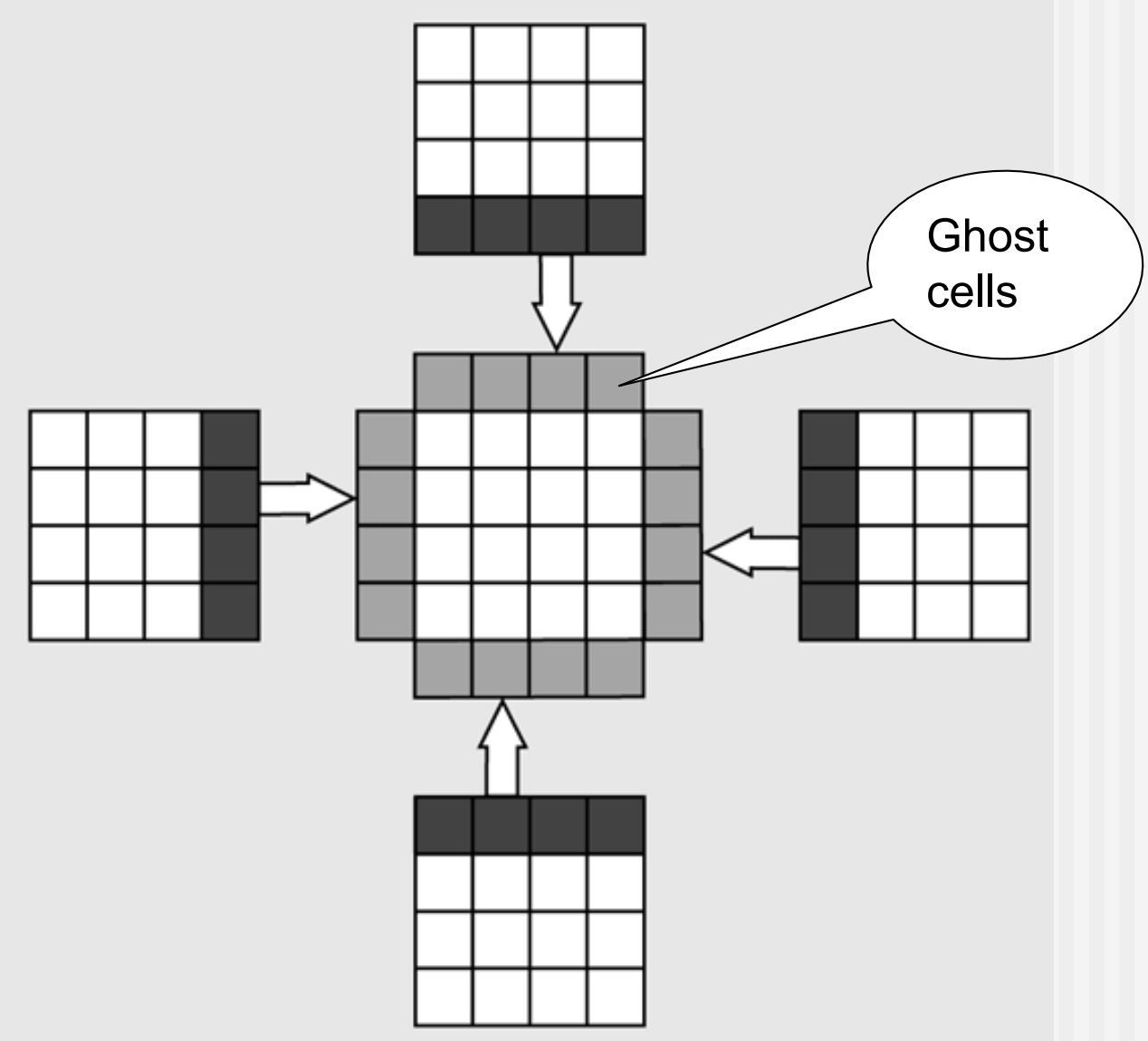
- Stencils are a typical access pattern

```
… = … a[i-1]+a[i]+a[i+1]
```



- Replicate overlap area or communicate it early on to avoid excessive communication inside loop

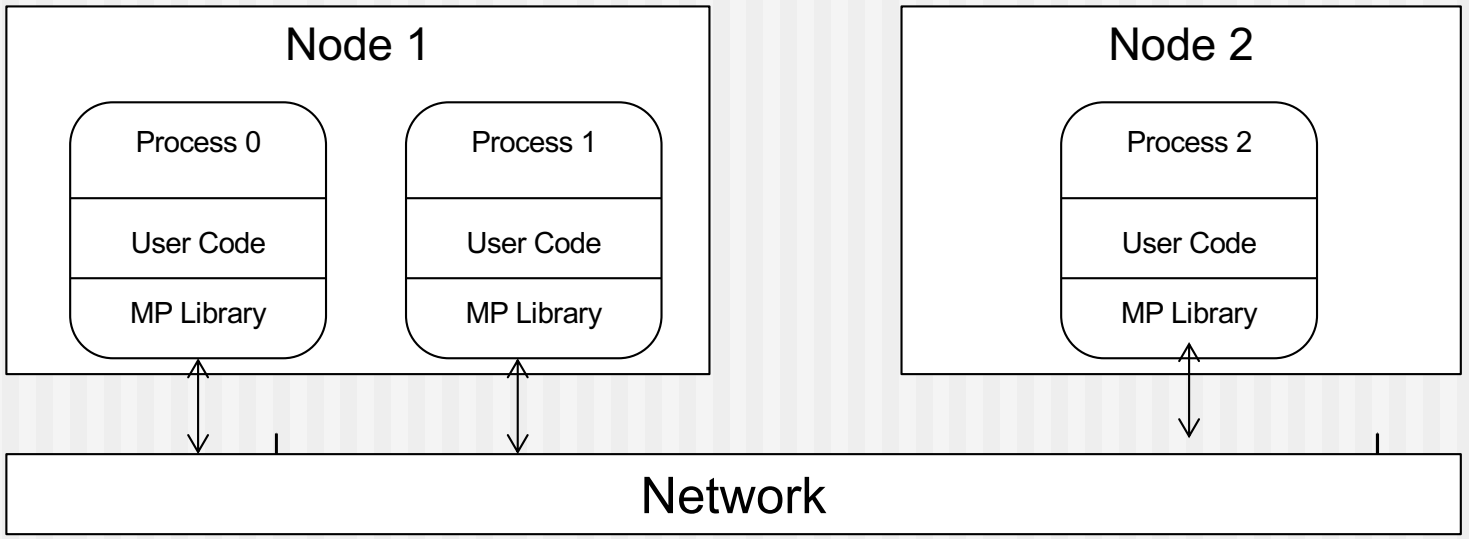
2D Overlap Area



Programming Distributed Memory Systems

Message Passing

- Different processes execute in different address space
  - In most cases on different cores or nodes
- Inter process communication by exchange of messages over the interconnection network
- Typically facilitated by library calls from within user program



Drawback of Threads and MP

- Threads and message passing are low level programming models
- It's the responsibility of the programmer to parallelize, synchronize, exchange messages
- Rather difficult to use

- Ideally we would like to have a parallelizing compiler that takes a standard sequential program and transforms it automatically into an efficient parallel program
  - In practice static compiler analysis cannot detect enough parallelism due to conservative treatment of dependencies

Parallel Languages

- Explicit parallel constructs
  - Parallel loops, array operations, …
  - Fortran >90, DPC/Sycl
- Compiler directives
  - “Hints” to the compiler on how to parallelize a program
  - OpenMP
- Directives are typically interpreted as comments by sequential compilers
  - Allows to compile parallel program with sequential compiler
  - Eases parallelization of legacy applications
- Partitioned Global Address Space (PGAS)

Attention

- Distributed Memory programming models can often also be applied to shared memory
  - Parallel languages:
    - Runtime system based on message passing or threads
    - Compiler support
  - Message passing
    - Use shared memory to do message passing - typically involves extra copies due to distributed address space of different processes

MPI – Basic Concepts

Erwin Laure

Director Max Planck Computing and Data Facility
& Technical University Munich

What is MPI

- **M P I** = **M**essage **P**assing **Interface**
- MPI is not an **implementation** – it is a **specification**
  - Specifies the interface of the library
- Interface specifications have been defined for C (C++) and Fortran programs.

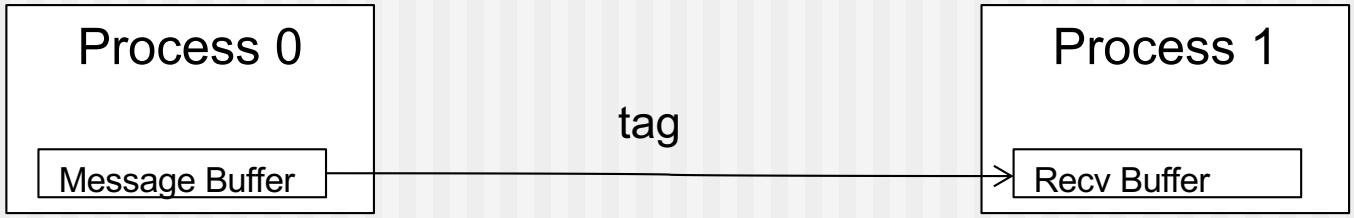
- Commonly used implementations of MPI:
  - MPICH (Argonne)
  - MVAPICH
  - OpenMPI
  - Vendor specific
    - Cray/HPE
    - Intel
    - IBM

A basic MP library

send(address, length, destination, tag)

- **address**: memory location signifying the beginning of the buffer containing the data to be sent,
- **length**: is the length in bytes of the message,
- **destination**: is the receiving process identifier
- **tag**: arbitrary integer to restrict receipt of message

recv (address, maxlen, source, tag, actlen)



Message Buffers

- (**address, length**) is insufficient in case of non-contiguous data and the need of data conversion
- MPI introduces datatypes
  - Basic datatypes predefined (MPI_INT, MPI_DOUBLE, …)
  - User can define own (non-contiguous) data types
- A message buffer in MPI is described as

```
(buf, count, datatype)
```

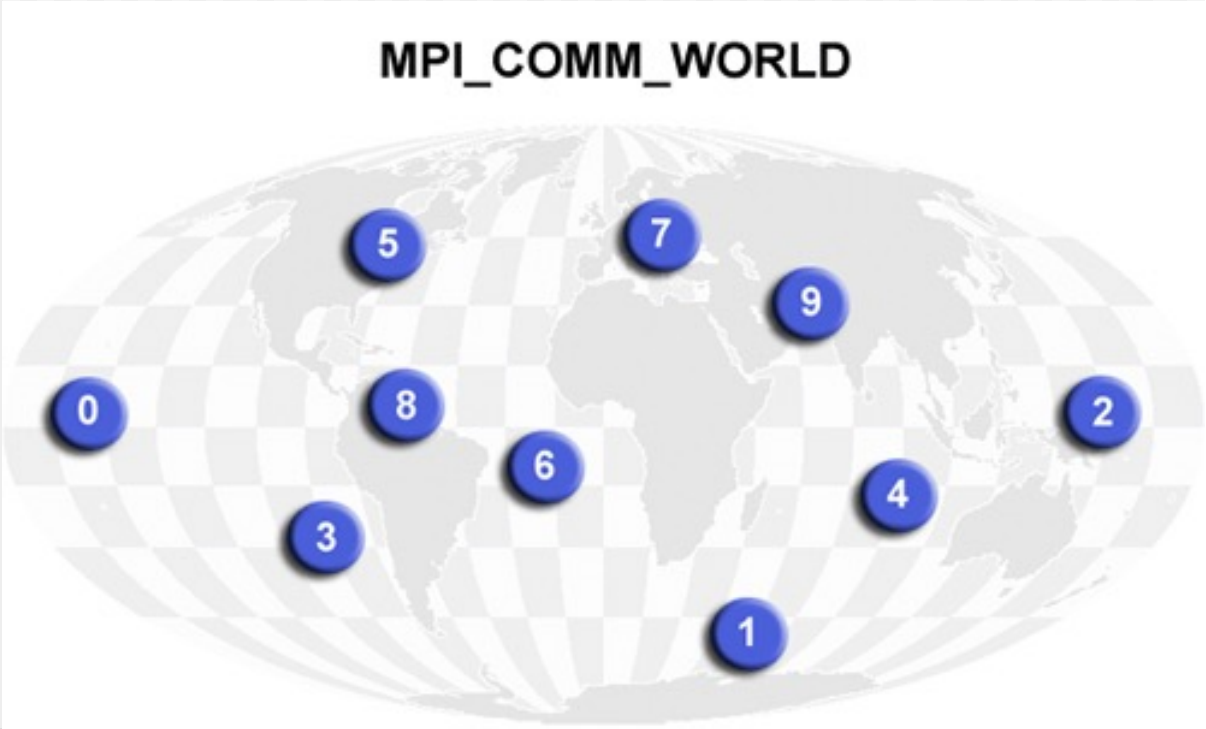
MPI Basic Datatypes (Fortran)

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Note: the names of the MPI C datatypes are slightly different

Processes and Communicators

- Processes belong to **groups**
- Processes within a group are identified with their **rank**
  - A group of n processes has ranks 0 … n-1
- MPI uses objects called **communicators** and groups to define which collection of processes may communicate with each other
  - `MPI_COMM_WORLD` is the default communicator covering all of the original MPI processes



Why Communicators?

- How to chose safe (unique) tags when writing a library?
  I.e. how to avoid a message being picked up by the wrong receiver?
- Collective operations (broadcast, reductions) can be easily defined over subgroups by using communicators
- Basis for advanced functionalities (mesh & graph topologies, neighbor communications, …)

Note: Processes vs. Processors

- MPI defines **processes**, it does not specify how these processes are mapped to physical **processors/cores**
- The mapping of processes to processors/cores is done at program start and dependent on the startup mechanism available on a certain resource – more about that later on.
- In principle, a MPI process does not necessarily correspond to an OS process – in practice it very often does.

Send/Receive in MPI

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

- `(buf, count, datatype)` describes the data to be sent
- `Dest` is the rank of the destination in the group associated with communicator `comm`
- `tag` is an identifier of the message
- `comm` identifies a group of processes

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

- `status` provides information on the message received, including source, tag, and count

Recap: Basic MPI Concepts

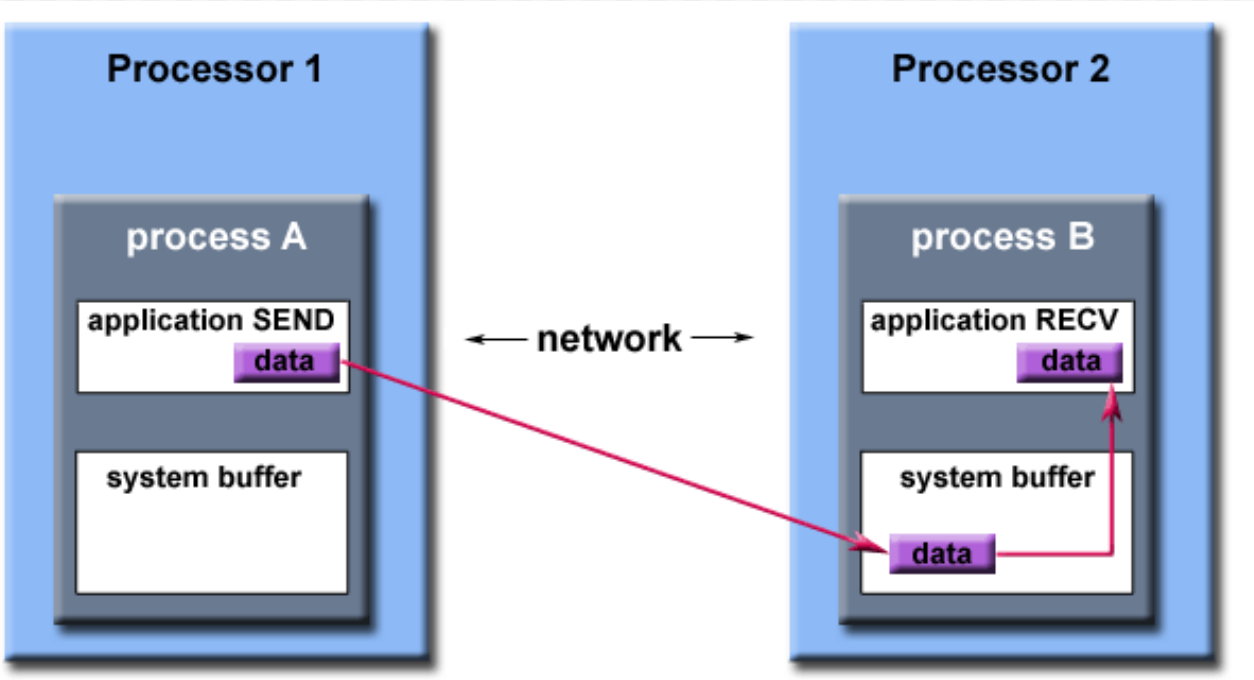
- Message **buffers** described by address, data type, and count
- Processes identified by their **ranks**
- **Communicators** identifying communication contexts/groups

MPI 4 Standard has over 1000 pages with several hundred functions …

- How many years do I have to study before I can use it?
- In fact, you will hardly ever use most of the MPI functions
- 6 functions are sufficient for simple programs:
  - `MPI_Init` – to initialize the MPI environment
  - `MPI_Comm_Size` – to know the number of processes
  - `MPI_Comm_Rank` – to know the rank of the calling process
  - `MPI_Send` – to send a message
  - `MPI_Recv` – to receive a message
  - `MPI_Finalize` – to exit in a clean way

What is not specified

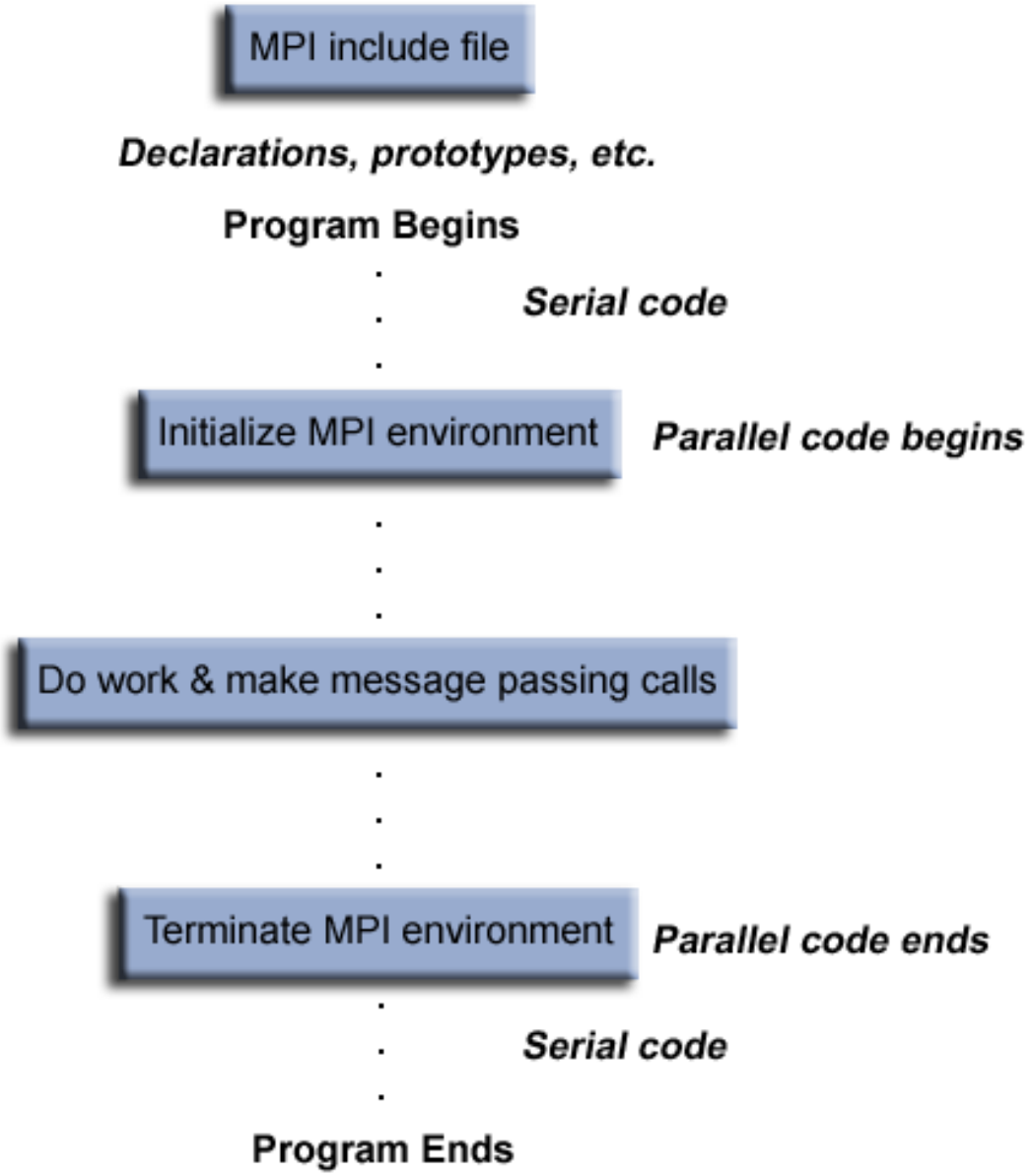
- Certain aspects are not specified in the MPI standard but left as implementation detail:
  - Process startup (how to start an MPI program)
    - All what happens before `MPI_Init` is executed
  - Richer error codes are allowed
  - Message buffering



Path of a message buffered at the receiving process

A first MPI Program

MPI Program Structure




```
#include "mpi.h"

rc = MPI_Init(&argc,&argv);

MPI_Comm_size(MPI_COMM_WORLD,&
numtasks);

MPI_Comm_rank(MPI_COMM_WORLD,&
rank);

MPI_Finalize();
```

Format of MPI Routines

- C Binding:
  - `rc = MPI_Xxxxx`(parameter, ...)
  - Example: `rc = MPI_Send(&buf,count,type,dest,tag,comm)`
  - Error code: Returned as "`rc`". `MPI_SUCCESS` if successful

- Fortran Binding
  - `call mpi_xxxxx(parameter,..., ierr)`
  - Example: `CALL MPI_SEND(buf,count,type,dest,tag,comm,ierr)`
  - Error code: Returned as "`ierr`" parameter. `MPI_SUCCESS` if successful

Example: Hello, World (C)

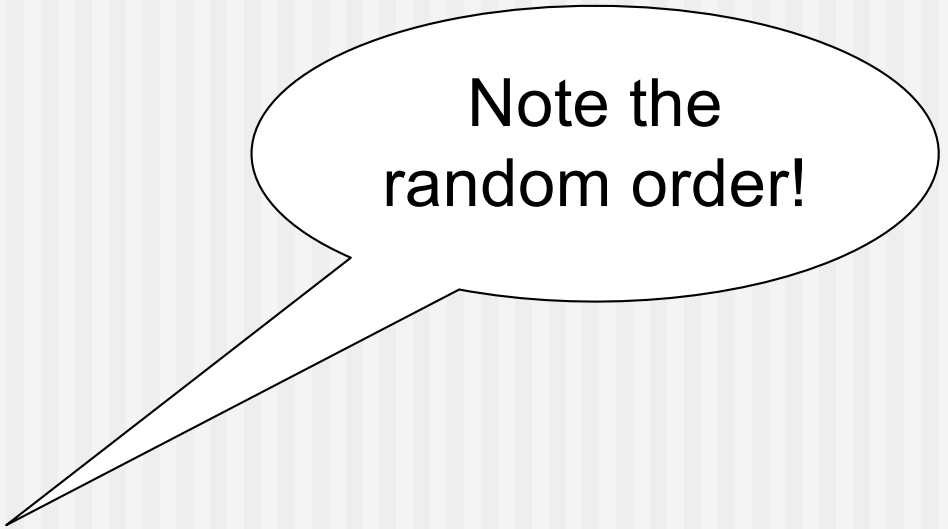
```
#include "mpi.h"
#include <stdio.h>

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

rc = MPI_Init(&argc,&argv);
if (rc != MPI_SUCCESS) {
  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 ("Hello, World from rank %d out of %d\n", rank, numtasks);
MPI_Finalize();
}
```

Example: Hello, World (Fortran)

```
program simple
include 'mpif.h'

integer numtasks, rank, ierr, rc

call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) then
   print *,'Error starting MPI program. Terminating.'
   call MPI_ABORT(MPI_COMM_WORLD, rc, ierr)
end if

call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
print *, 'Hello, World from rank ',rank, ' out of=',numtasks

call MPI_FINALIZE(ierr)

end
```

Sample Output (24 processes)

```
Hello, World from rank 9 out of 24
Hello, World from rank 17 out of 24
Hello, World from rank 13 out of 24
Hello, World from rank 7 out of 24
Hello, World from rank 11 out of 24
Hello, World from rank 14 out of 24
Hello, World from rank 16 out of 24
Hello, World from rank 4 out of 24
Hello, World from rank 15 out of 24
Hello, World from rank 3 out of 24
Hello, World from rank 23 out of 24
Hello, World from rank 10 out of 24
Hello, World from rank 5 out of 24
Hello, World from rank 12 out of 24
Hello, World from rank 2 out of 24
Hello, World from rank 19 out of 24
Hello, World from rank 21 out of 24
Hello, World from rank 8 out of 24
Hello, World from rank 18 out of 24
Hello, World from rank 1 out of 24
Hello, World from rank 6 out of 24
Hello, World from rank 22 out of 24
Hello, World from rank 20 out of 24
Hello, World from rank 0 out of 24
```



How to launch MPI Programs?

- Not specified by MPI standard
- Many implementations use `mpirun -np X`
  - Hostfile used to specify processes/hardware mapping
- MPI standard proposes, but does not mandate, a common `mpiexec` syntax/semantics, similar to `mpirun`
- Dardel uses `srun -n x`

Hands on

- Compile and run the hello world example
- Compiler:
  - cc
  - ftn
- Request interactive resources
  - salloc -N 1 -A edu24.summer -t 0:10:00 –p lab-08-22 (23)
- Run
  - `srun –n 16 a.out`
- `Code`
  - `hello_mpi.c/f90`

Summary

- MPI Basics
  - Message buffers
  - Processes and communicators
  - Structure of MPI programs
  - Implementation specific features
- To find out the exact syntax of certain commands:
  - On Dardel use `> man MPI_xxx`
  - Look up Web resources

Basic MPI
Point-to-Point Communication

Erwin Laure

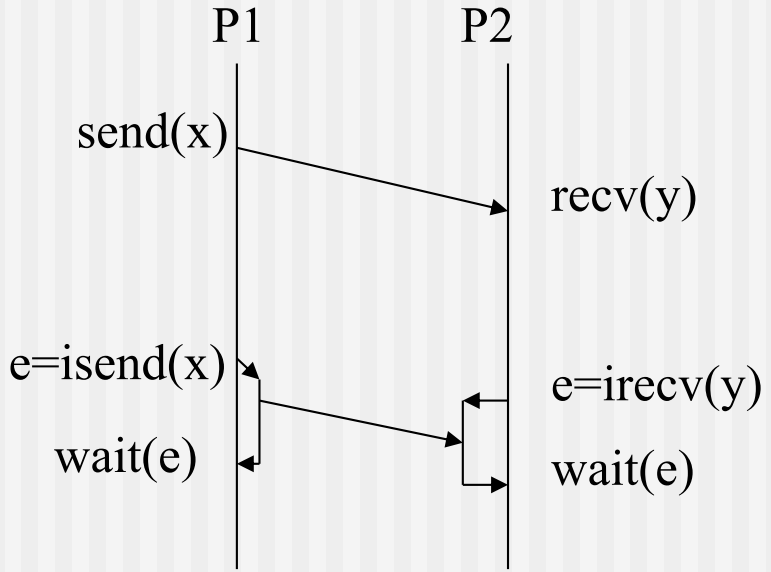
Director Max Planck Computing and Data Facility
& Technical University Munich

Contents

- Sending data from A to B
  - Message format
  - Buffers and semantics
  - Communication modes
- Deadlocks
- Blocking and non-blocking communication

Sending Data from A to B …

- The basic function of any message passing library
  - Typically a SEND/RECEIVE pair
- Needed when process X needs data from process Y
- Two main incarnations
  - Blocking: stops the program until it is safe to continue
  - Non-blocking: separates communication from computation



Send/Receive in MPI

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

- `(buf, count, datatype)` describes the data to be sent
- `Dest` is the rank of the destination in the group associated with communicator `comm`
- `tag` is an identifier of the message
- `comm` identifies a group of processes

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

- `status` provides information on the message received, including source, tag, and count

Basic MPI Message Syntax

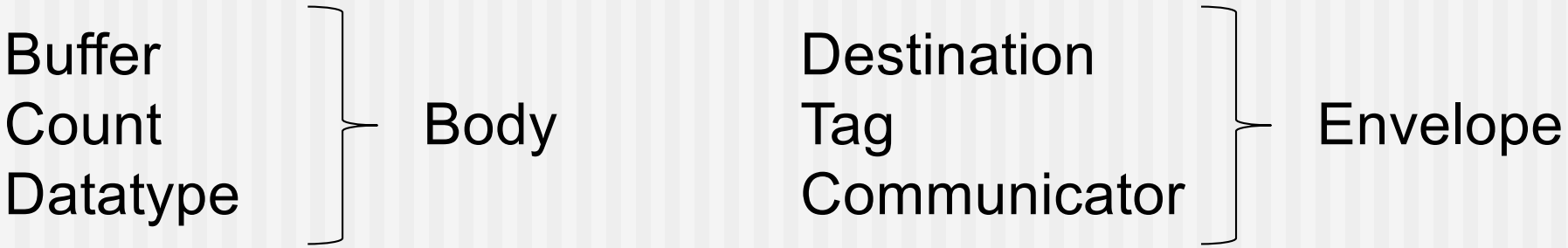
- An MPI message consists of an **envelope** and **message body** – think of it like a letter in the mail:

- The **envelope** of an MPI message has four parts:
  - **Source** — the sending process
  - **Destination** — the receiving process
  - **Communicator** — specifies a group of processes to which both source and destination belong
  - **Tag** — used to classify messages

- The **message body** has three parts:
  - **Buffer** — the message data
  - **Datatype** — the type of the message data
  - **Count** — the number of items of type datatype in buffer

Basic Send/Receive Commands

```
int MPI_Send(void *buf, int count, MPI_Datatype
dtype, int dest, int tag, MPI_Comm comm);

MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, IERR)
```

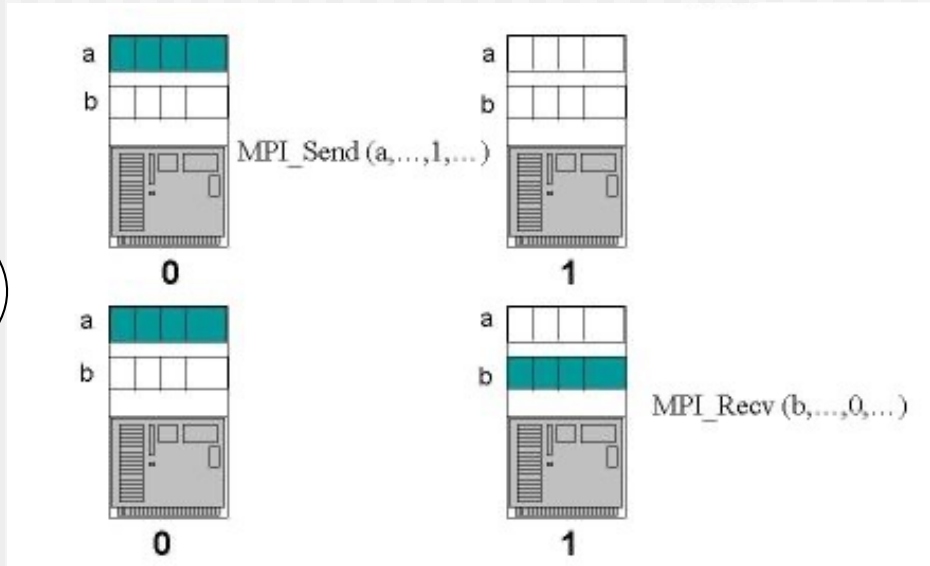



```
int MPI_Recv(void *buf, int count, MPI_Datatype
dtype, int source, int tag, MPI_Comm comm, MPI_Status
*status);

MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,
STATUS, IERR)
```

Example

```
double a[100],b[100];

if( myrank == 0 )          /* Send a message */
{
  for (i=0;i<100;++i)
     a[i]=sqrt(i);
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
}
else if( myrank == 1 )    /* Receive a message */
  MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD, &status );
```

What happens on P0 if `b` is replaced with `a`?



Wildcards

- Instead of specifying everything in the envelope explicitly, wildcards can be used for sender and tag:

  `MPI_ANY_SOURCE` and `MPI_ANY_TAG`

- Actual source and tag are stored in `STATUS` variable

C:

```
MPI_Status status;
MPI_Recv(b, 100, MPI_DOUBLE,
         MPI_ANY_SOURCE, MPI_ANY_TAG,
         MPI_COMM_WORLD, &status );

source = status.MPI_SOURCE;
tag = status.MPI_TAG;
```

Wildcards cont'd

- Fortran:

```
integer status(MPI_STATUS_SIZE)
call MPI_RECV(b, 100, MPI_DOUBLE_PRECISON,
          MPI_ANY_SOURCE, MPI_ANY_TAG,
          MPI_COMM_WORLD, status, ierr );

tag = status(MPI_TAG)
source = status(MPI_SOURCE)
```

Message Size

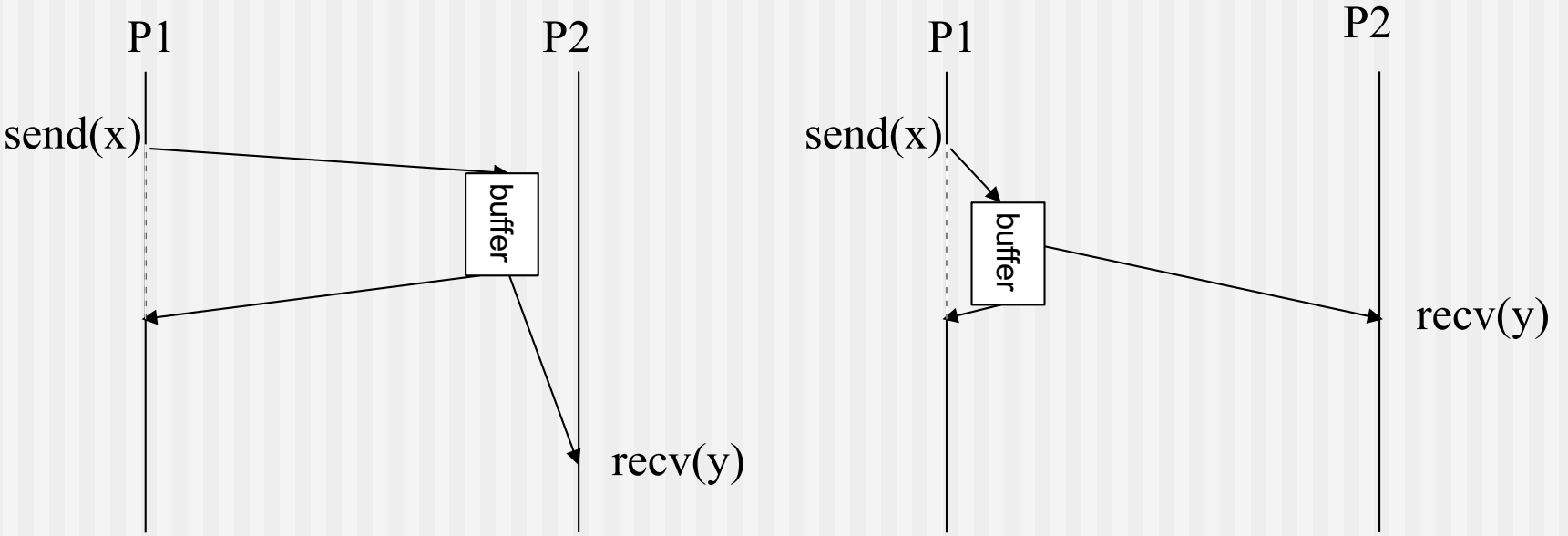
- Semantics of receiving buffer is that it has to be at least as large as the message to be received – the actual data received might be smaller!

- Again, actual information is stored in `STATUS` variable:

```
int MPI_Get_count(MPI_Status *status,
      MPI_Datatype dtype, int *count);
```

A Word on Buffering

- MPI implementations typically use (**internal**) message buffers
  - Sending process can safely modify the sent data once it is copied into the buffer, irrespectively of status of receiving process
  - Receiving process can buffer incoming messages even if no (user space) receiving buffer is provided, yet
  - Buffers can be on both sides



Note

This system buffer is **DIFFERENT** to the message buffer you specify in the `MPI_Send` or `MPI_Recv` calls!

A Word on Buffering Cont'd

- The efficiency of MPI implementations critically depends on how buffers are being handled
  - A great source for optimization
  - Out of scope for this lecture

- Different handling of buffers can show different effects – hard to debug!
  - E.g. while in general no handshake between sending and receiving process is needed (i.e sending process may continue after data is copied into buffer even if no matching receive has been posted, yet) large messages or lack of buffering space may require synchronization with receiving process
  - No handshake is often called "**eager protocol**", handshake "**rendezvous protocol**"
  - Sometimes explicit buffers are required (see later) and lack of sufficient buffer space will cause the communication to fail.

Blocking and Completion

- Both `MPI_Send` and `MPI_Recv` are blocking
  - They program only continues after they are completed
- The command is completed once it is safe to (re)use the data
  - `MPI_Recv`: data has been fully received
  - `MPI_Send`: can be completed even if no non-local action has been taking place. WHY?
  - Once data is copied into a send buffer `MPI_Send` can complete

Hands on

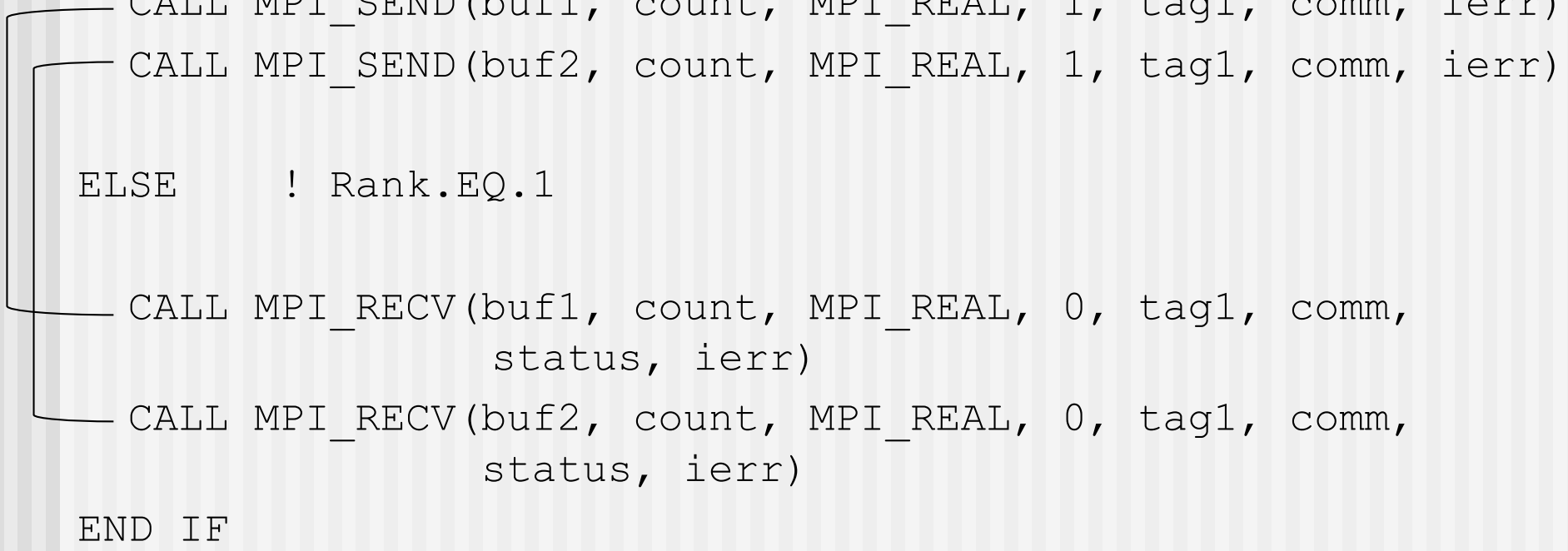
- Propagate data through all processes
  - process 0 sends to process 1
  - process n receives from process n-1 and sends to n+1
- Modify the code such that process 0 sends data to all others
- Code: send_recv.c/f90

Hands on – Approximate Pi

- The given PI program calculates PI using an integral approximation. Take the serial version of the program and modify it to run in parallel.
- First familiarize yourself with the way the serial program works. How does it calculate PI?
- Hint: look at the program comments. How does the precision of the calculation depend on DARTS and ROUNDS, the number of approximation steps?
- Hint: edit DARTS to have various input values from 10 to 10000. What do you think will happen to the precision with which we calculate PI when we split up the work among the nodes?
- Now parallelize the serial PI program. Use only the six basic MPI calls.
- Hint: As the number of darts and rounds is hard coded then all workers already know it, but each worker should calculate how many are in its share of the DARTS so it does its share of the work. When done, each worker sends its partial sum back to the master, which receives them and calculates the final sum.
- Code: pi_serial.c/f90
- What are the differences between receiving from a specified worker (i.e. loop index) and using MPI_ANY_SOURCE?

Message Order

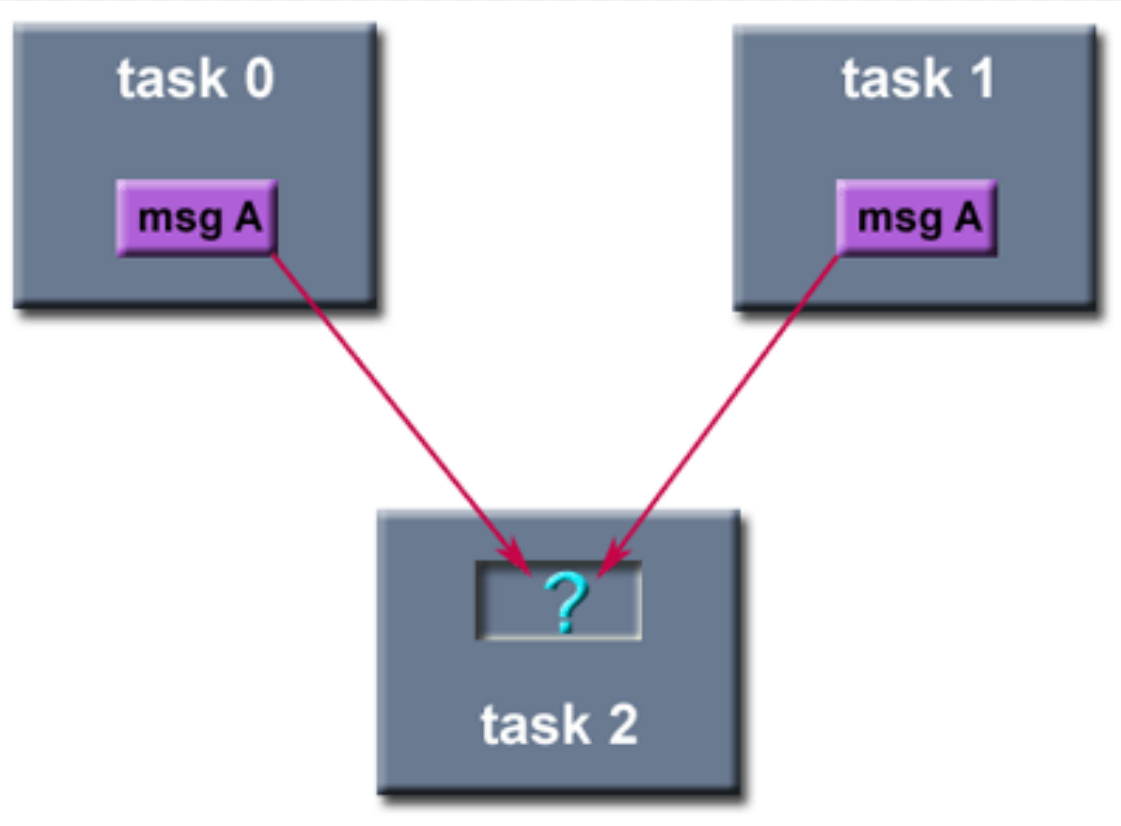
- MPI messages are non-overtaking
  - If the sender sends two messages (with the same envelope) to the same destination they have to be received in the same order



```
IF (rank.EQ.0) THEN
   CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
   CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag1, comm, ierr)

ELSE    ! Rank.EQ.1

   CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag1, comm,
                 status, ierr)
   CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm,
                 status, ierr)
END IF
```

Fairness

- MPI makes no guarantees about fairness
  - If there are two matching sends (from different sources) for a receive any of these can be successful
  - MPI does not prevent operation starvation (e.g. sends that will never be picked up)



What have we learned?

- The semantics of `MPI_Send/MPI_Recv` are quite implementation dependent

- How can we get more control on what is actually happening?
  - MPI provides different communication modes with different semantics

MPI Communication Modes

- Synchronous mode
  - Syntax: `MPI_Ssend(…)`
  - Semantics: handshake required, send will block until matching receive has been posted and receiving has started
- Ready mode
  - Syntax: `MPI_Rsend(…)`
  - Semantics: user guarantees that matching receive has already been posted; similar to synchronous but no need for handshake
- Buffered mode
  - Syntax: `MPI_Bsend(…)`
  - Semantics: send buffer will be used and command returns once data is locally copied; send buffer needs to be provided by user

Discussion

- Standard `MPI_Send(…)` behaves like `MPI_Bsend` or `MPI_Ssend` depending on message size and internal buffer space
- For portability and safety reasons you should always assume `MPI_Ssend` semantics
  - Don’t assume `MPI_Send(…)` will return irrespectively of matching receive status

Discussion Cont'd

- `MPI_Bsend` will fail if not enough buffer space is available
  - You must provide sufficient buffer space in user space to an MPI process:

```
int MPI_Buffer_attach( void* buffer, int size)
MPI_BUFFER_ATTACH( BUFFER, SIZE, IERROR)

int MPI_Buffer_detach( void* buffer_addr, int* size)
MPI_BUFFER_DETACH( BUFFER_ADDR, SIZE, IERROR)
```

- This buffer is only used for buffered send and detach will block until all data is actually sent.

Pros and Cons of different modes

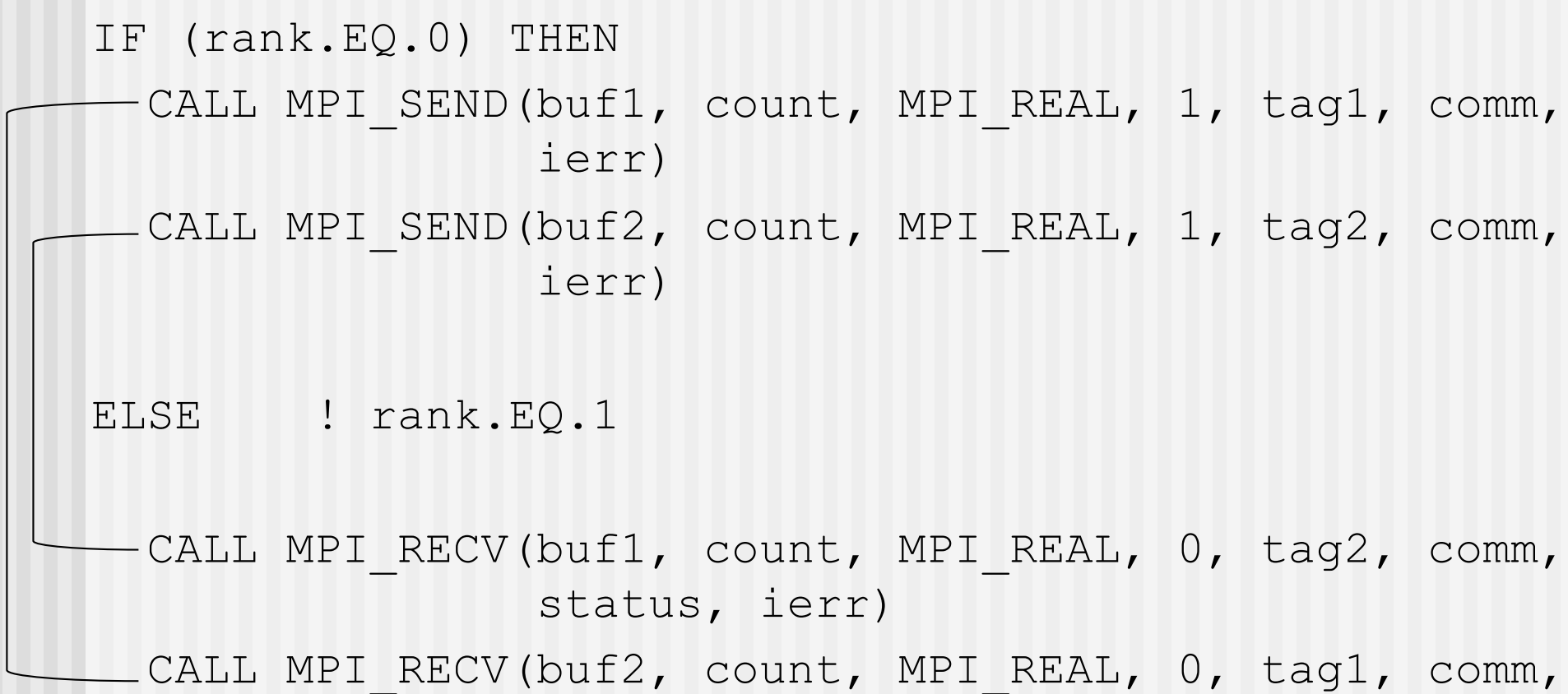
Advantages	Disadvantages
Synchronous Mode	
Safest, most portable	Can occur substantial synchronization overhead
Ready Mode	
Lowest total overhead	Difficult to guarantee that receive precedes send
Buffered Mode	
Decouples send from receive	Potentially substantial overhead through buffering
Standard Mode	
Most flexible, general purpose	Implementation dependent

Deadlocks

- Deadlocks are common (and hard to debug) errors in message passing programs
- A deadlock occurs when two (or more) processes wait on the progress of each other:

```
if( myrank == 0 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
              &status );
    MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  }
  else if( myrank == 1 ) {
    /* Receive, then send a message */
    MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
              &status );
    MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
```

Deadlock or not?



```
IF (rank.EQ.0) THEN
    CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm,
                  ierr)
    CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm,
                  ierr)

ELSE    ! rank.EQ.1

    CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm,
                  status, ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm,
                  status, ierr)

END IF
```

How to avoid Deadlocks?

- Careful organize the communication in your program
  - Make sure sends are always paired with receives in the correct order
  - A difficult task in large programs!
- Don’t depend on how specific implementations handle their internal buffers
  - A program may work well with certain problem sizes but deadlock once you increase the problem size or move to a different architecture or MPI implementation because of internal buffer limitations

Communication modes revisited


```
IF (rank.EQ.0) THEN
  CALL MPI_SSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```


```
IF (rank.EQ.0) THEN
  CALL MPI_SEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```


```
IF (rank.EQ.0) THEN
  CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
  CALL MPI_SEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
ELSE    ! rank.EQ.1
  CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
  CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
END IF
```

Help to avoid Deadlock

- Careful ordering of send/receives is facilitated by a combined send/receive command:

```
int MPI_Sendrecv( void *sendbuf, int sendcount,
                  MPI_Datatype sendtype,
                  int dest, int sendtag,
                  void *recvbuf, int recvcount,
                  MPI_Datatype recvtype,
                  int source, int recvtag, MPI_Comm
                  comm, MPI_Status *status )
```

- Advantage: order of send/receive irrelevant; receive will not be blocked by potentially blocking send
- Very useful for shift operations

Sendrcv Example

```
if (myid == 0) then
   call mpi_send(a,1,mpi_real,1,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(b,1,mpi_real,1,tag,MPI_COMM_WORLD,
                 status,ierr)
elseif (myid == 1) then
   call mpi_send(b,1,mpi_real,0,tag,MPI_COMM_WORLD,ierr)
   call mpi_recv(a,1,mpi_real,0,tag,MPI_COMM_WORLD,
                 status,ierr)
end if

if (myid == 0) then
   call mpi_sendrecv(a,1,mpi_real,1,tag1,
                     b,1,mpi_real,1,tag2,
                     MPI_COMM_WORLD, status,ierr)
elseif (myid == 1) then
   call mpi_sendrecv(b,1,mpi_real,0,tag2,
                     a,1,mpi_real,0,tag1,
                     MPI_COMM_WORLD, status,ierr)
end if
```

Help to avoid Deadlocks Cont'd

- Careful message ordering
  - Always a good idea!
- Buffered communication
  - But comes with (quite substantial) overhead
- Non-blocking calls

Non-blocking Communication

- For all send/receive calls there is a non-blocking equivalent named `I(x)send/Irecv`
- Non-blocking calls will return immediately irrespectively of the send/receive status
  - They actually only **initiate** the action
  - Actual sending/receiving of messages will be handled internally in the MPI implementation
  - Calls return a handle that allows to check the progress of sending/receiving
- Blocking and non-blocking calls can be intermixed
  - A blocking receive can match a non-blocking send and vice-versa.

Non-blocking Syntax

```
int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int
dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int
source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_ISEND(BUF, COUNT, DTYPE, DEST, TAG, COMM, REQ, IERR)
MPI_IRECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM, REQ, IERR)
```

- Request is the handle to the request
- **Important**: None of the arguments passed to a non-blocking send/recv **must** be written or read until the send/recv operation is completed.

Completion of non-blocking send/receives

```
int MPI_Wait( MPI_Request *request, MPI_Status
*status );
MPI_WAIT(REQUEST, STATUS, IERR )
```

- `MPI_Wait` is blocking and will only return when the message has been sent/received
  - After `MPI_Wait` returns it is safe to access the data again

```
int MPI_Test( MPI_Request *request, int *flag,
              MPI_Status *status );
MPI_TEST(REQUEST, FLAG, STATUS, IERR)
```

- MPI_Test returns immediately
  - Status of request is returned in flag (true for done, false when still ongoing)

Deadlock Example revisited

```
if( myrank == 0 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
 }
 else if( myrank == 1 ) {
   /* Receive, then send a message */
   MPI_Recv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &status );
   MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
```

Example

```
if( myrank == 0 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 1, 19, MPI_COMM_WORLD,
             &request );
  MPI_Send( a, 100, MPI_DOUBLE, 1, 17, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
else if( myrank == 1 ) {
  /* Post a receive, send a message, then wait */
  MPI_Irecv( b, 100, MPI_DOUBLE, 0, 17, MPI_COMM_WORLD,
             &request );
  MPI_Send( a, 100, MPI_DOUBLE, 0, 19, MPI_COMM_WORLD );
  MPI_Wait( &request, &status );
}
```

- No deadlock because non-blocking receive is posted before send

Discussion

- Non-blocking communication has two main benefits:
  - Helps avoid deadlocks
  - Allows to overlap communication with computation (latency hiding)
    - More about that later on
- Disadvantage:
  - Makes code more complex to read/understand and thus debug and maintain.
  - Limitations of internal data structures to keep track of outstanding requests

Summary

- MPI provides blocking and non-blocking communication
- 4 communication modes

- You should now be able to program message passing applications
- Everything you want to do can be done with the (6) basic commands you know now.
  - But many tasks would be awkward and inefficient – hence the lecture continues

- Beware deadlocks!

Basic MPI
Collective Communication

Erwin Laure

Director Max Planck Computing and Data Facility
& Technical University Munich

What we know already

- Everything to write MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication

Collective Communication

- Often more than 2 processes are involved in communication
  - Send input data to all processes
  - Collect results from all processes
  - Synchronize all processes
  - Update all processes with partial results
  - …
- All this can be implemented with the commands you already know
  - But it is tedious, error-prone, and difficult to implement efficiently
- Hence MPI provides ready-made commands for this

Collective Communication Cont'd

- Communication involving all processes in a **group** (i.e. a **communicator**)
  - MPI-3 defines "neighborhood collectives"
- All processes in a group **MUST** participate to the collective operation
- No tag mechanism, only order of program execution
  - Remember that MPI messages cannot overtake another one
- Until MPI-2 all collective routines were only blocking
  - With the standard completion semantics of blocking communication – thus no guarantee there is a full synchronization
  - MPI-3 introduced non-blocking collectives
    - Important difference to non-blocking p2p: no matching with non-blocking collectives!

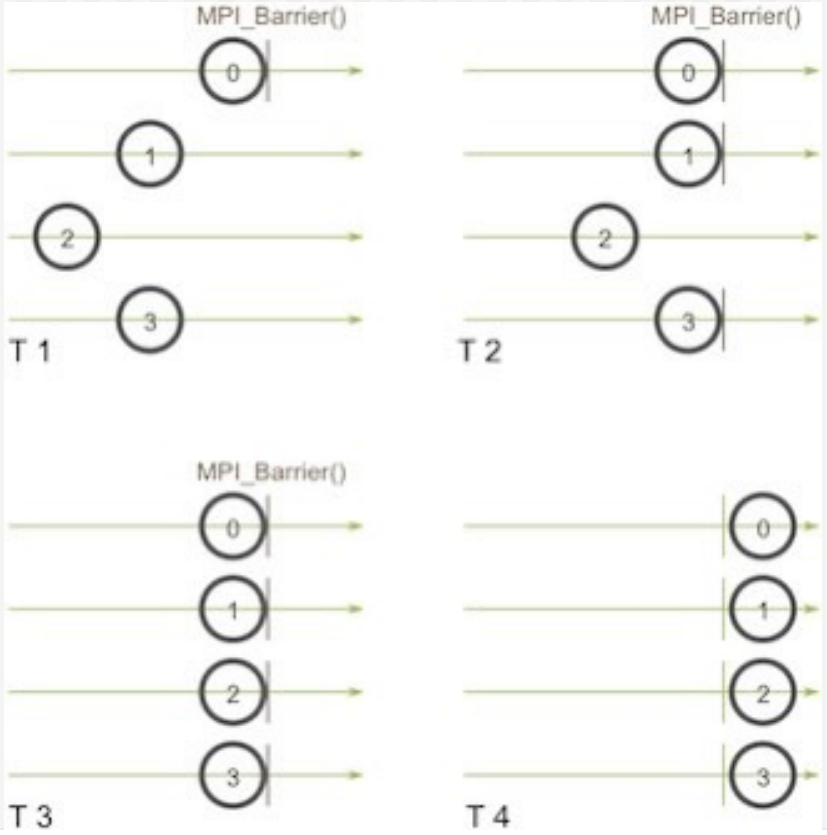
List of Collective Routines

- Barrier synchronization across all processes.
- Broadcast from one process to all other processes
- Global reduction operations such as sum, min, max or user-defined reductions
- Gather data from all processes to one process
- Scatter data from one process to all processes
- All-to-all exchange of data
- Scan across all processes

Barrier Synchronization

- Sometimes there is a need to synchronize all processes before them continuing independently
  - E.g. read in input data
- `MPI_Barrier` blocks the calling process until all processes in the group have also called `MPI_Barrier`

```
int MPI_Barrier (MPI_comm comm)

MPI_BARRIER (COMM, ERROR)
```

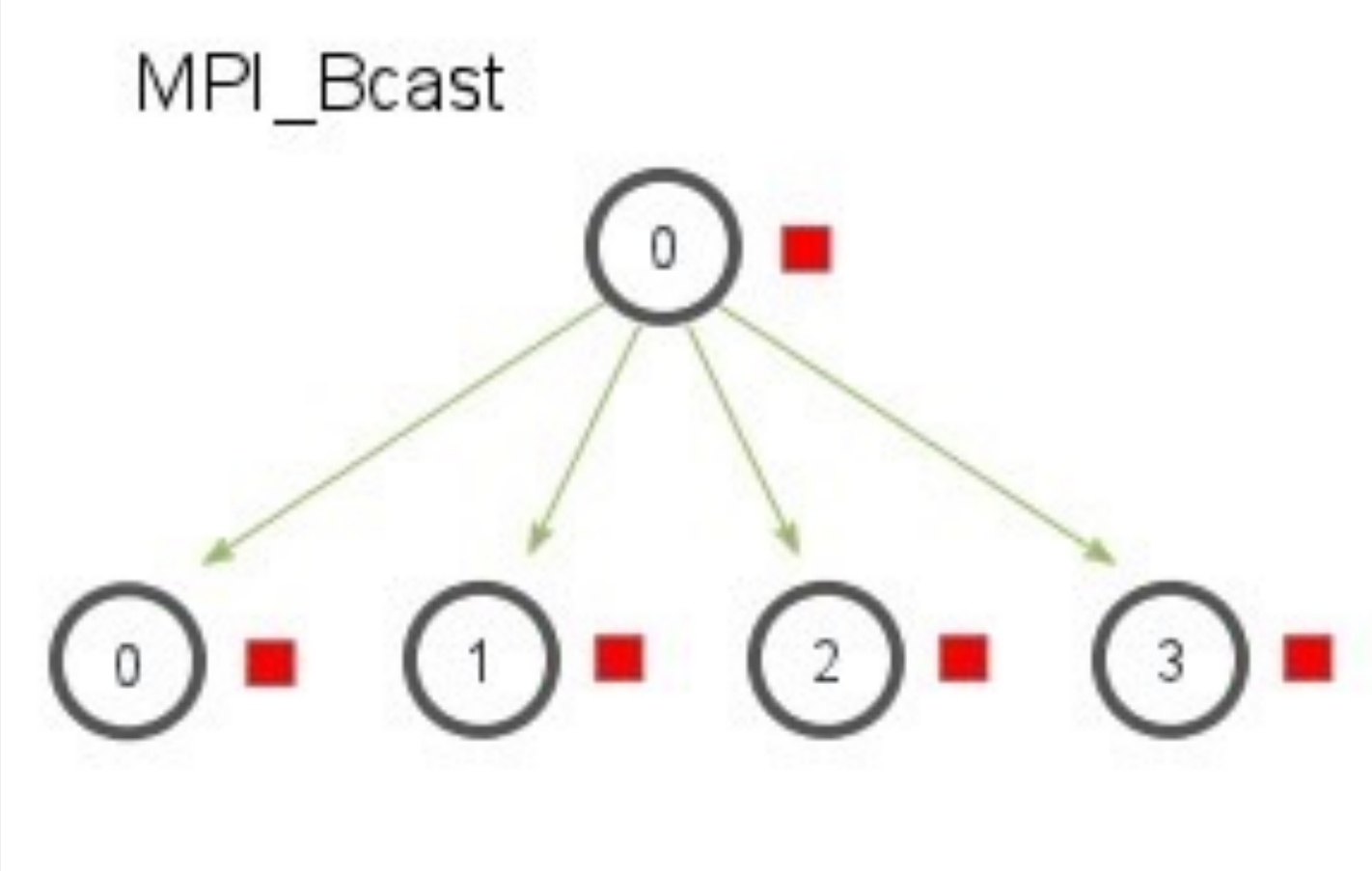


Hands on

- Use MPI_BARRIER to enforce consecutive ordering of output messages in hello_mpi.c/f90

Broadcast

- Broadcast sends data from one process to the same memory location in all other processes
  - send and receive buffer are the same!



Broadcast Cont'd

```
int MPI_Bcast (void* buffer, int count,
               MPI_Datatype datatype,
               int root, MPI_Comm comm )
MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT,
           COMM, IERR )
```

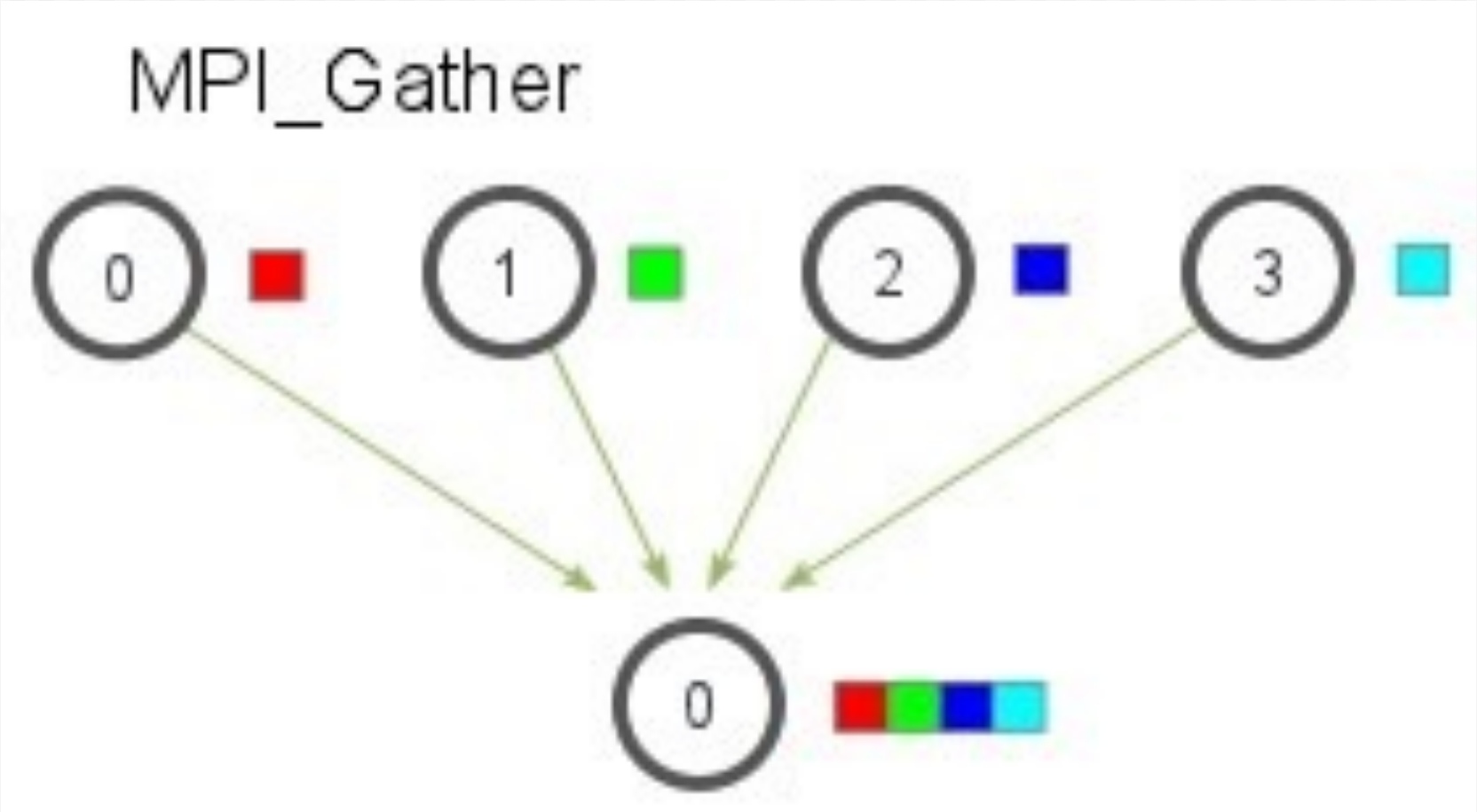
- Note:
  - Only one (send/receive) buffer
  - No tag
  - `Root` indicates the process owning the data to be broadcasted

Broadcast Example

```
#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();
}
```

Gather

- Gather is a all-to-one operation that collects the data from all processes in target process



Gather Cont'd

```
int MPI_Gather (void* send_buffer, int send_count,
                MPI_datatype send_type, void* recv_buffer,
                int recv_count, MPI_Datatype recv_type,
                int rank, MPI_Comm comm )

MPI_GATHER (SEND_BUFFER, SEND_COUNT, SEND_TYPE,RECV_BUFFER,
            RECV_COUNT, RECV_TYPE, RANK, COMM, ERROR )
```

- Note:
  - Each process (including the root process) sends the contents of its send buffer to the root process. The root process receives the messages and stores them in rank order.
  - Receive buffer needs to be large enough to store all data
  - The gather could also be accomplished by each process calling `MPI_SEND` and the root process calling `MPI_RECV` *N* times to receive all of the messages.
  - all processes, including the root, must send the **same** amount of data, and the data are of the same type.

Gather Example

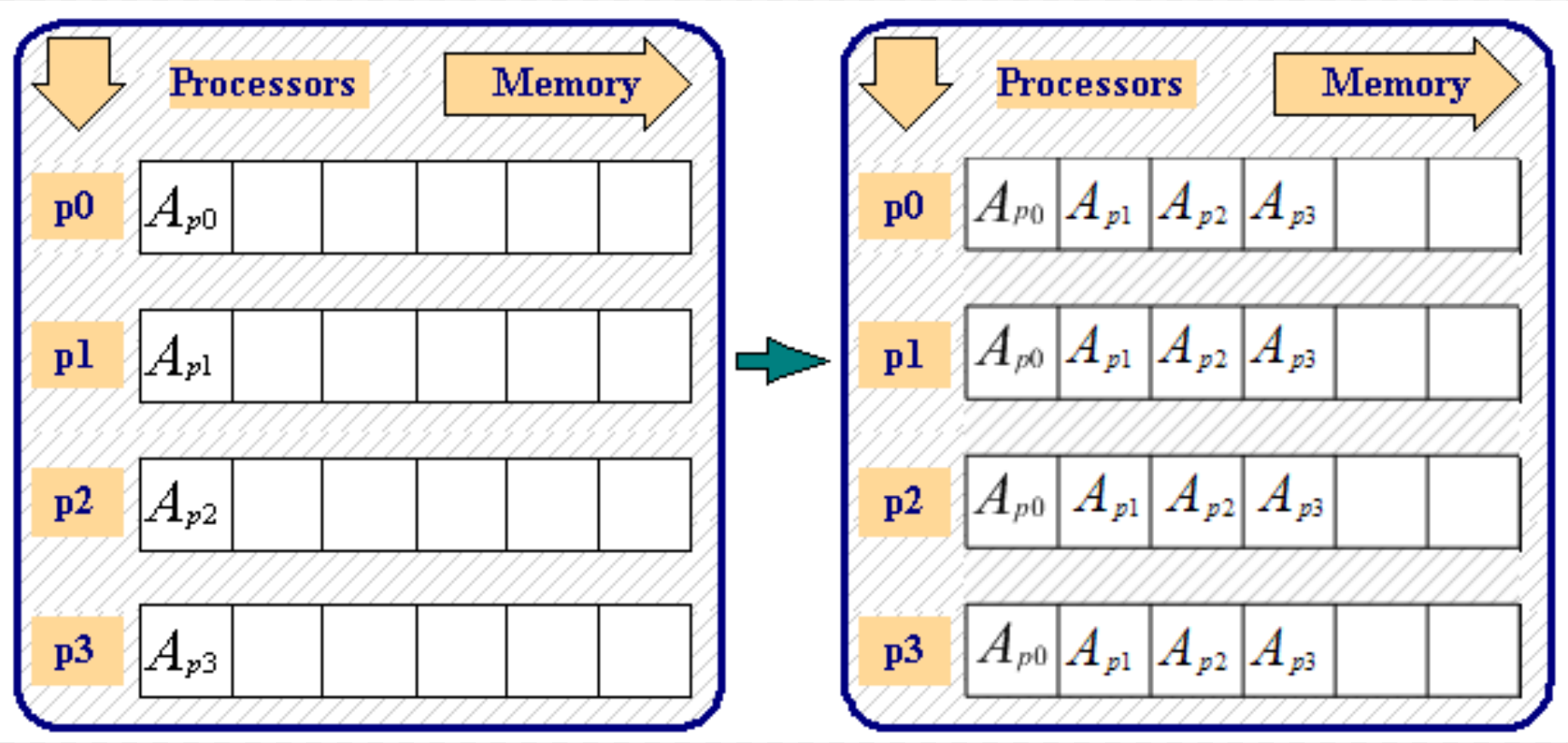
```
int rank,size;
double param[16],mine;
int sndcnt,rcvcnt; I;

sndcnt=1;
mine=23.0+rank;
if(rank==7) rcvcnt=1;

MPI_Gather(&mine,sndcnt,MPI_DOUBLE,param,rcvcnt,
           MPI_DOUBLE,7,MPI_COMM_WORLD);

if(rank==7)
for(i=0;i<size;++i) printf("PE:%d param[%d] is %f \n",
    rank,i,param[i]]);
```

Hands on

- Modify Pi_mpi.c/f90 to use MPI_GATHER on P0
- Hint: pirecv needs to turn into an array
- Hint: think about whether the calculation of pi_est needs to change

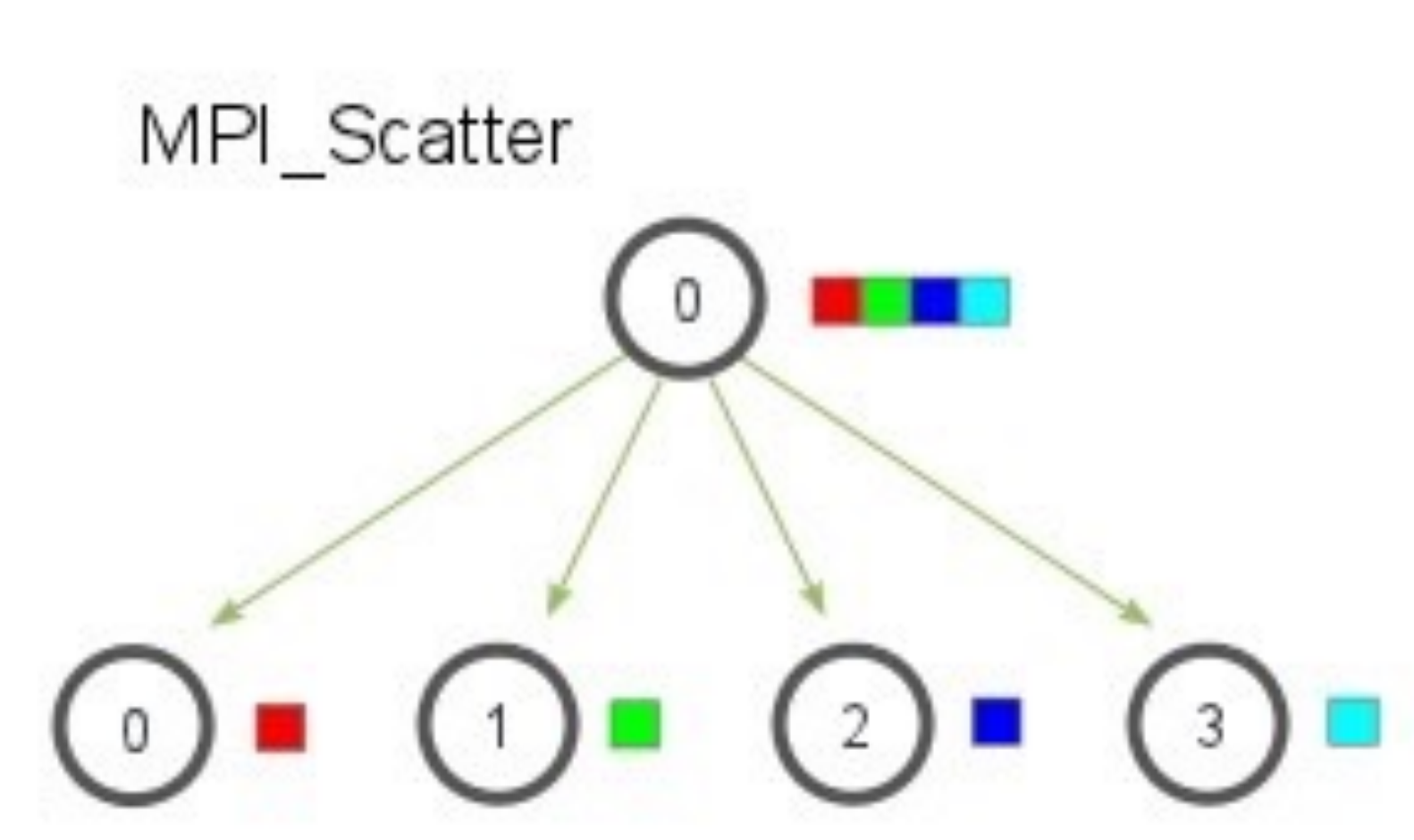
Allgather

- Sometimes it is also useful to gather the data not only into one process but all
- Equivalent to `MPI_Gather` plus `MPI_Bcast`
- `MPI_Allgather` has same syntax as `MPI_Gather`



Scatter

- Distribute data to all processes – one-to-all communication
- Inverse to gather



Scatter Cont'd

```
int MPI_Scatter (void* send_buffer, int send_count,
                 MPI_datatype send_type,
                 void* recv_buffer, int recv_count,
                 MPI_Datatype recv_type,
                 int rank, MPI_Comm comm )

MPI_Scatter (SEND_BUFFER, SEND_COUNT, SEND_TYPE,
             RECV_BUFFER, RECV_COUNT, RECV_TYPE,
             RANK, COMM, ERROR )
```

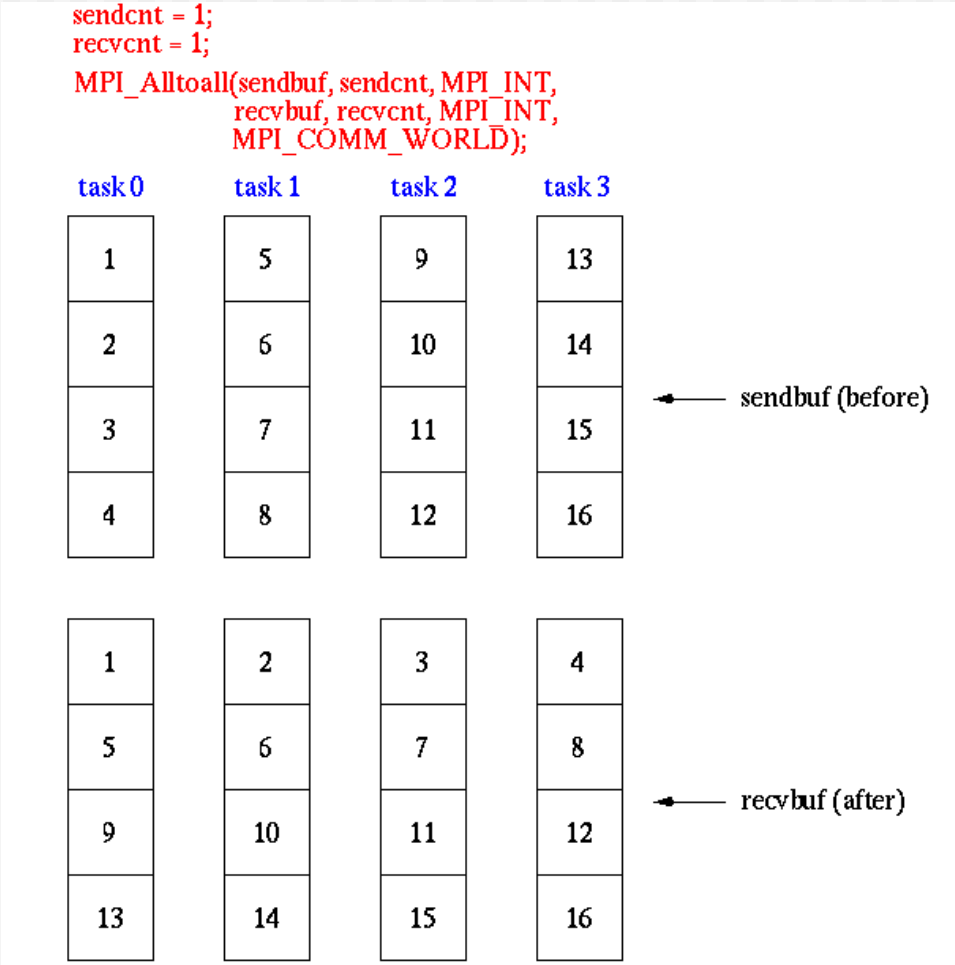
- root process breaks up the send buffer into equal chunks and sends one chunk to each processor.
  - The outcome is the same as if the root executed *N* MPI_SEND operations and each process executed an MPI_RECV.

Scatter Example

```
rcvcnt=1;
if(rank==3) {
  for(i=0;i<8;++i) param[i]=23.0+i;
  sndcnt=1;
}
MPI_Scatter(param,sndcnt,MPI_DOUBLE,&mine,rcvcnt,
            MPI_DOUBLE,3,MPI_COMM_WORLD);
for(i=0;i<size;++i)  {
  if(rank==i) printf("P:%d mine is %f \n",rank,mine);
  fflush(stdout);
  MPI_Barrier(MPI_COMM_WORLD);
}
MPI_Finalize();
}
```

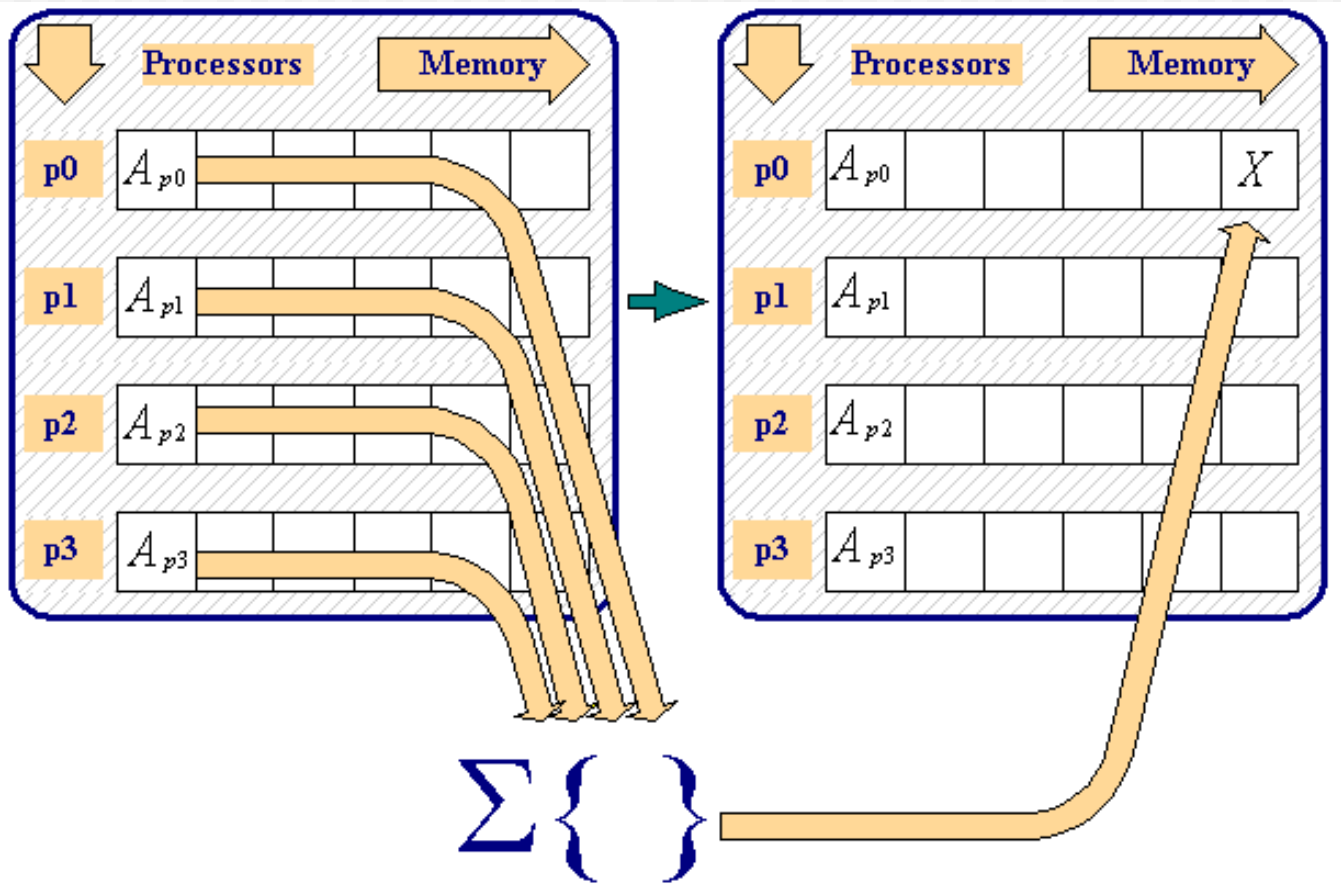
Other Gather/Scatter Variants

- Gather/Scatter is also defined over vectors
  - `MPI_GATHERV` and `MPI_SCATTERV` allow a varying count of data from/to each process.
- `MPI_ALLTOALL`
  - Every process performs a scatter



Reduction

- Collect data from each processor
- Reduce these data to a single value (such as a sum or max)
- Store the reduced result on the root processor



Reduction Cont'd

```
int MPI_Reduce (void* send_buffer, void* recv_buffer, int
                count, MPI_Datatype datatype, MPI_Op
                operation, int rank, MPI_Comm comm )

MPI_REDUCE ( SEND_BUFFER, RECV_BUFFER, COUNT, DATATYPE,
             OPERATION, RANK, COMM, ERROR )
```

- Note:
  - `Rank` denotes the process that stores the result in `recv_buffer`
  - `Operation` can be one of 12 pre-defined operations or user-defined
  - Both send and receive buffers must have the same number of elements with the same type.
    - The arguments `count` and `datatype` must have identical values in all processes.
  - The argument `rank` must also be the same in all processes.

Predefined Reduction Operations

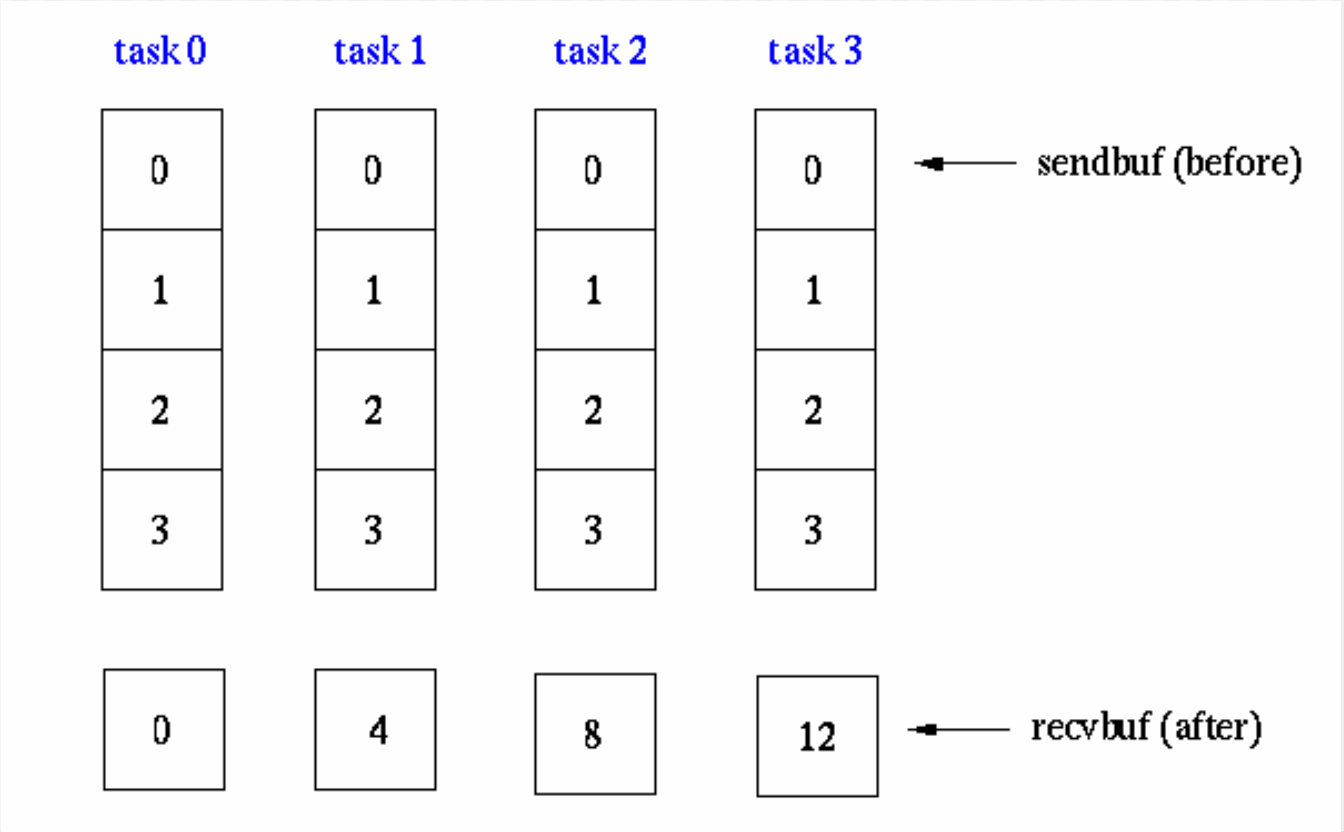
Operation	Description
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bitwise xor
MPI_MINLOC	computes a global minimum and an index attached to the minimum value -- can be used to determine the rank of the process containing the minimum value
MPI_MAXLOC	computes a global maximum and an index attached to the rank of the process containing the maximum value

Reduction Example

```
#include   <stdio.h>
#include   <mpi.h>
void main(int argc, char *argv[]) {
  int rank;
  int source,result,root;

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

  root=7;
  source=rank+1;

  MPI_Reduce(&source,&result,1, MPI_INT, MPI_PROD, root,
             MPI_COMM_WORLD);
  if(rank==root) printf("P:%d MPI_PROD result is %d \n", rank,
                        result);

MPI_Finalize();
}
```

Reduce Variations

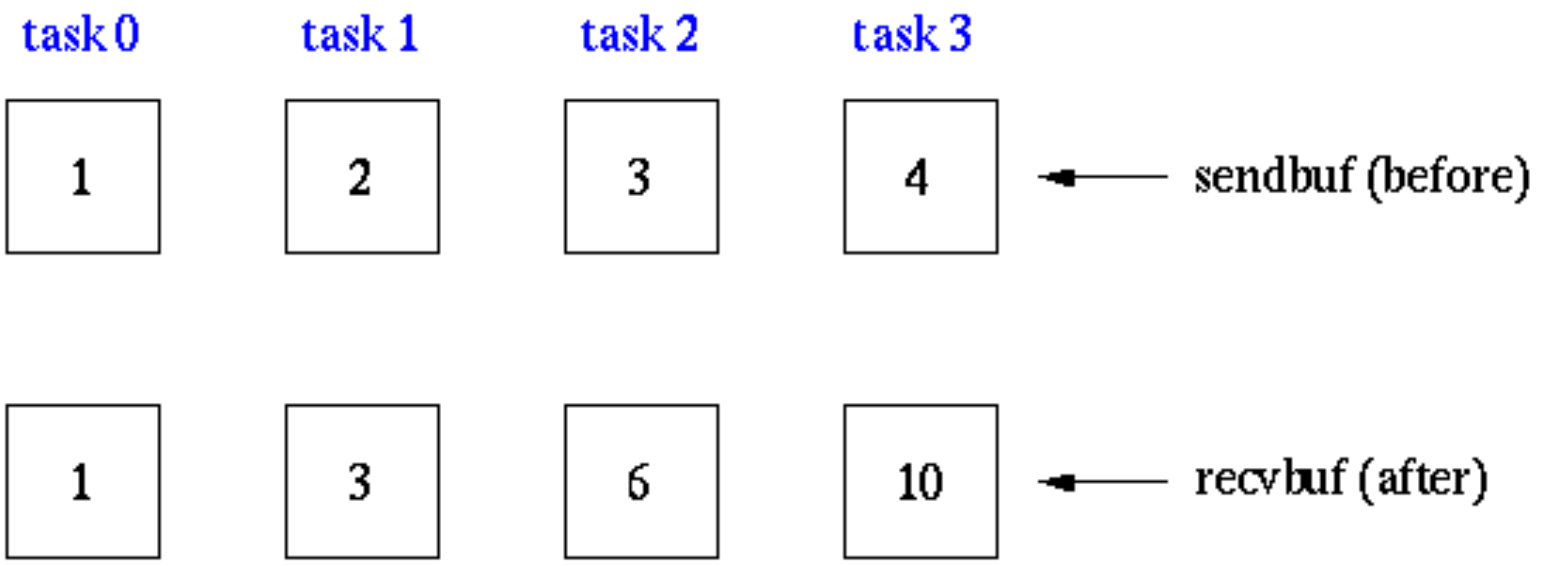
- `MPI_Allreduce` makes the result available in the receive buffers of all processes
  - Equivalent to `MPI_Reduce` plus `MPI_Bcast`
- `MPI_Reduce_scatter` scatters the result vector across the processes in the group



Reduce Variations Cont'd

- `MPI_Scan` performs a partial reduction in which process i receives data from processes 0 through i, inclusive

```
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT, MPI_SUM,
         MPI_COMM_WORLD);
```



Hands on

- Modify Pi_mpi.c/f90 to use MPI_REDUCE

Summary

- Collective communication routines provide convenient calls for standard communication patterns
- Depending on the implementation they may be much more efficient than hand-coding (or not)
  - Synchronization overhead might be substantial
- Collective communication makes extensive use of groups/communicators

What's next

- Intermediate MPI
  - Overlapping communication/computation
  - Using communicators
  - Derived datatypes

Intermediate MPI

Erwin Laure

Director Max Planck Computing and Data Facility
& Technical University Munich

What we know already

- Everything to write MPI programs
  - Program structure
  - Point-to-point communication
  - Communication modes
  - Blocking/non-blocking communication
  - Collective Communication

Take a deeper look

- Usage of data types
  - So far we used the pre-defined data types; what if we need to deal with more complex structures?
- Usage of communicators
  - How to group processes in individual groups
- Improving Communication Performance
  - Aka how to speed up programs

Recap: MPI Datatypes

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE_PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Note: the names of the MPI C datatypes are slightly different

Derived Datatypes

- Primitive datatypes are contiguous (basically arrays)
- Derived Datatypes allow you to define your own data structures based upon sequences of the MPI primitive data types.
- Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.
- MPI provides several methods for constructing derived data types:
  - Contiguous
  - Vector
  - Indexed
  - Struct

Example

- Send one row of a matrix:
  - Data is contiguous in C; can simply send
  - But it is not contiguous in Fortran

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

- Send one column of a matrix:
  - Same as above but contiguous in Fortran

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

- How to solve non-contiguous case?
  - Send each element in separate message
    - Overhead and error prone

Send contiguous data

- Could be achieved simply with

```
MPI_Send(&a[i][0], 4, MPI_FLOAT, j, tag,
         MPI_COMM_WORLD);
```

- If you do this frequently, you might want to use a more descriptive datatype name (eg. coordinate point) and help MPI packing the data

Equivalent to above

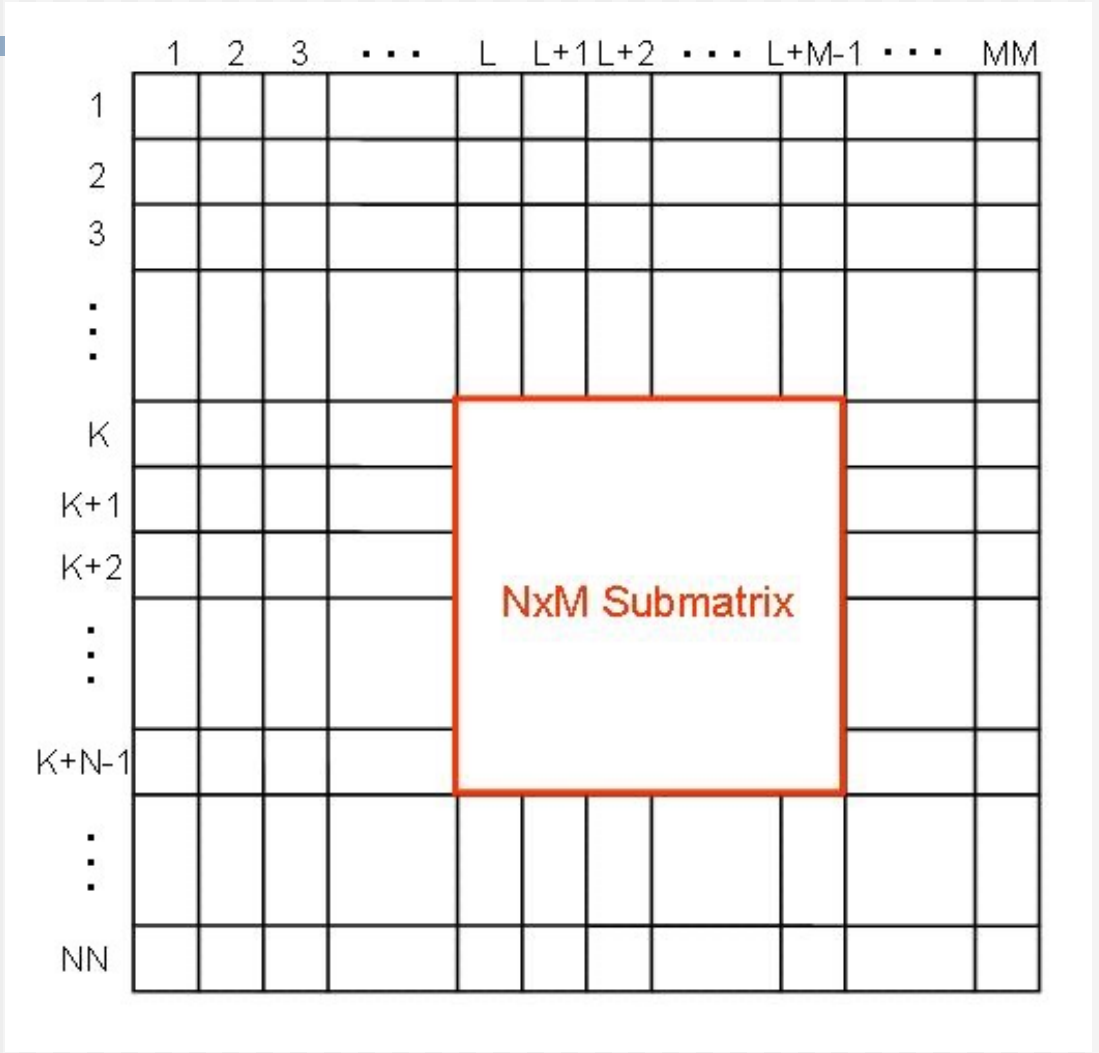
```
MPI_Type_contiguous(4, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

MPI_Send(&a[i][0], 1, rowtype, j, tag,
         MPI_COMM_WORLD)
```

Example Cont'd

```
MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);

if (numtasks == SIZE) {
  if (rank == 0) {
     for (i=0; i<numtasks; i++)
       MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
     }

  MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD,
&stat);
  printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
         rank,b[0],b[1],b[2],b[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);
```

- Note different type in send/recv
- Is the program safe?

Example: submatrix




```
do j = 1, m
   call MPI_Send(a(k,l+j-1), n, MPI_DOUBLE,
        dest, tag, MPI_COMM_WORLD, ierr)
enddo
```

First Approach: Buffering

- Create a user-level buffer for the sub-matrix:

```
icount = 0
do j = l, l+m-1
   do i = k, k+n-1
      icount = icount + 1
      p(icount) = a(i,j)
   enddo
enddo

call MPI_Send(p, n*m, MPI_DOUBLE, dest, tag,
              MPI_COMM_WORLD, ierr)
```

- Limitations:
  - Usage of memory and CPU time to do buffering
  - Still can use only one datatype in the buffer
  - Need to interpret the buffer correctly on the receiving side

A better Approach: Derived Datatypes

- `MPI_TYPE_Vector`: Similar to contiguous, but allows for regular gaps (stride) in the displacements

```
call MPI_TYPE_VECTOR(m, n, nn, MPI_DOUBLE,
                     my_mpi_type, ierr)
call MPI_TYPE_COMMIT(my_mpi_type, ierr)
call MPI_SEND(a(k,l), 1, my_mpi_type, dest, tag,
              MPI_COMM_WORLD, ierr)
```

- `m`…count (we send m columns)
- `n`…number of contiguous elements (each column has n elements)
- `nn`…stride (distance between the starting locations of adjacent blocks of data. The columns of the full matrix each have *NN* values, so *NN* will be the stride between the beginning of one column segment and an adjacent column segment.)

Different Derived Datatypes

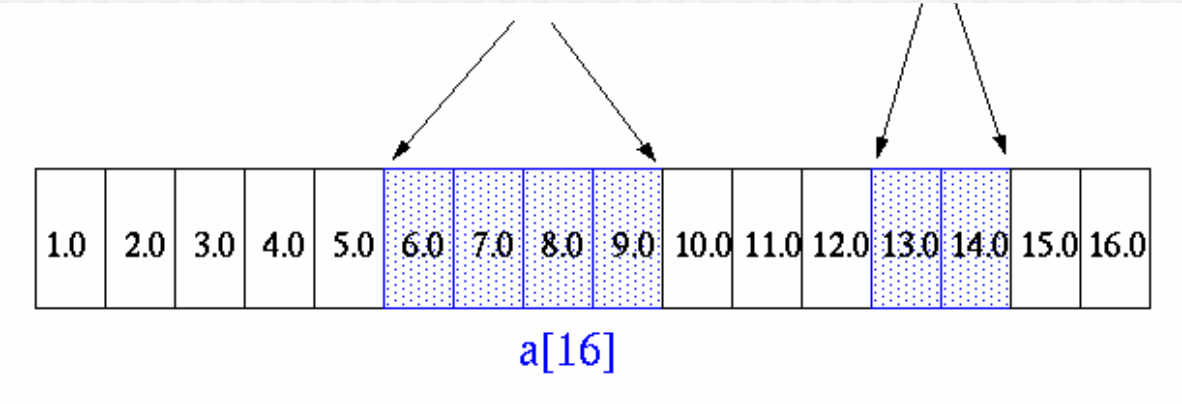
- **Contiguous**: This is the simplest constructor. It produces a new datatype by making count copies of an existing one.
- **Vector**: This is a slight generalization of the contiguous type that allows for regular gaps in the displacements. Elements are separated by multiples of the extent of the input datatype.
- **Hvector**: This is like vector, but elements are separated by a specified number of bytes.
- **Indexed** and **Hindexed**: An array of displacements of the input datatype is provided; the displacements are measured in terms of the extent of the input datatype or in bytes.
- **Struct**: This provides a fully general description.

Indexed

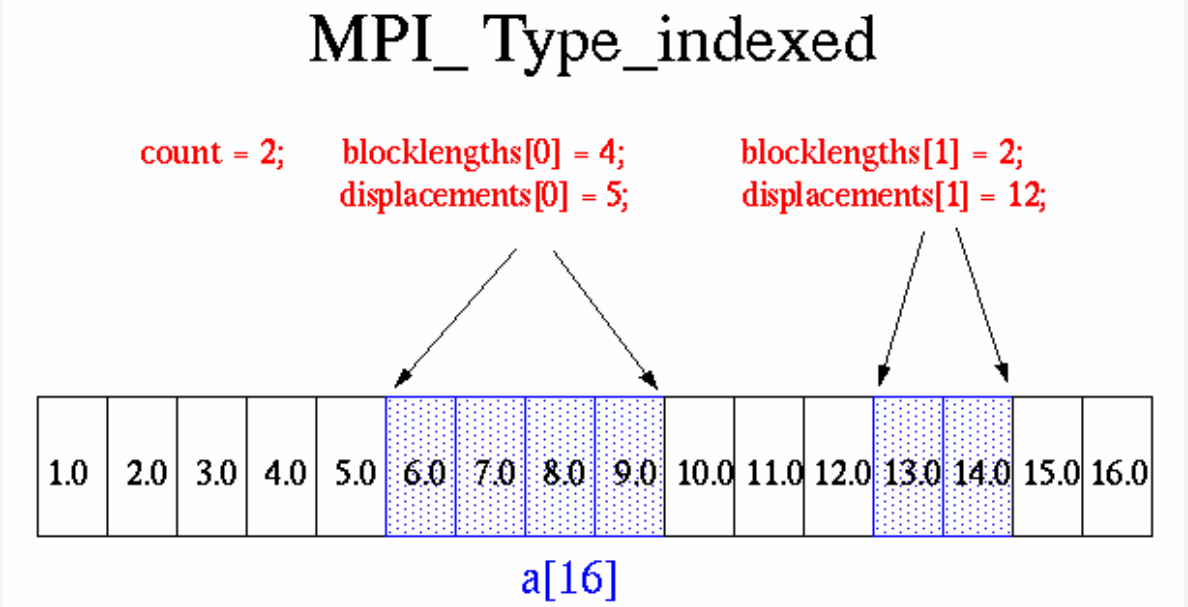
```
int MPI_Type_indexed(int count,
     const int *array_of_blocklengths,
     const int *array_of_displacements,
     MPI_Datatype oldtype,
     MPI_Datatype *newtype);

Input Parameters:
* count: number of blocks - also number of entries in
       array_of_displacements and array_of_blocklengths
* array_of_blocklengths: number of elements in each  block
       (array of nonnegative integers)
* array_of_displacements: displacement of each block in
       multiples of oldtype (array of integers) - always
       from beginning
* oldtype: old datatype (handle)

Output Parameters
* newtype: new datatype (handle)
```

Hands On

- Send elements 6-9 and 13-14 of array a from rank 0 to rank1
- source files: indexed.f90 or indexed.c

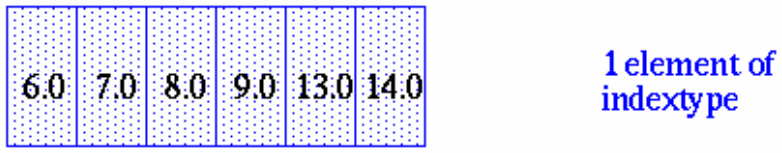


Solution




```
MPI_Type_indexed(count, blocklengths, displacements, MPI_FLOAT, &indextype);
```

```
MPI_Send(&a, 1, indextype, dest, tag, comm);
```



Struct

```
int MPI_Type_create_struct(
        int count,
        int array_of_blocklengths[],
        MPI_Aint array_of_displacements[],
        MPI_Datatype array_of_types[],
        MPI_Datatype *newtype );
```

Struct Example

```
Struct Particlestruct{
  double x,y,z,velocity;
  int n,type;
} particle[100];

MPI_Datatype particletype;
MPI_Datatype type[2] = {MPI_DOUBLE,
                        MPI_INT};
Int blocklen[2] = {4,2};
MPI_Aint disp[2] = {0, 4*sizeof(double)};

MPI_Type_create_struct(2, blocklen, disp, type,
&particletype);
MPI_Type_commit(&particletype);

MPI_Send(particle, 100, particletype, dest, tag,
comm);
```

Derived Datatypes Summary

- MPI allows to create user defined datatypes
- Useful if non-contiguous memory locations need to be communicated
- The created derived datatype should be used frequently in a program – otherwise overhead might be too large

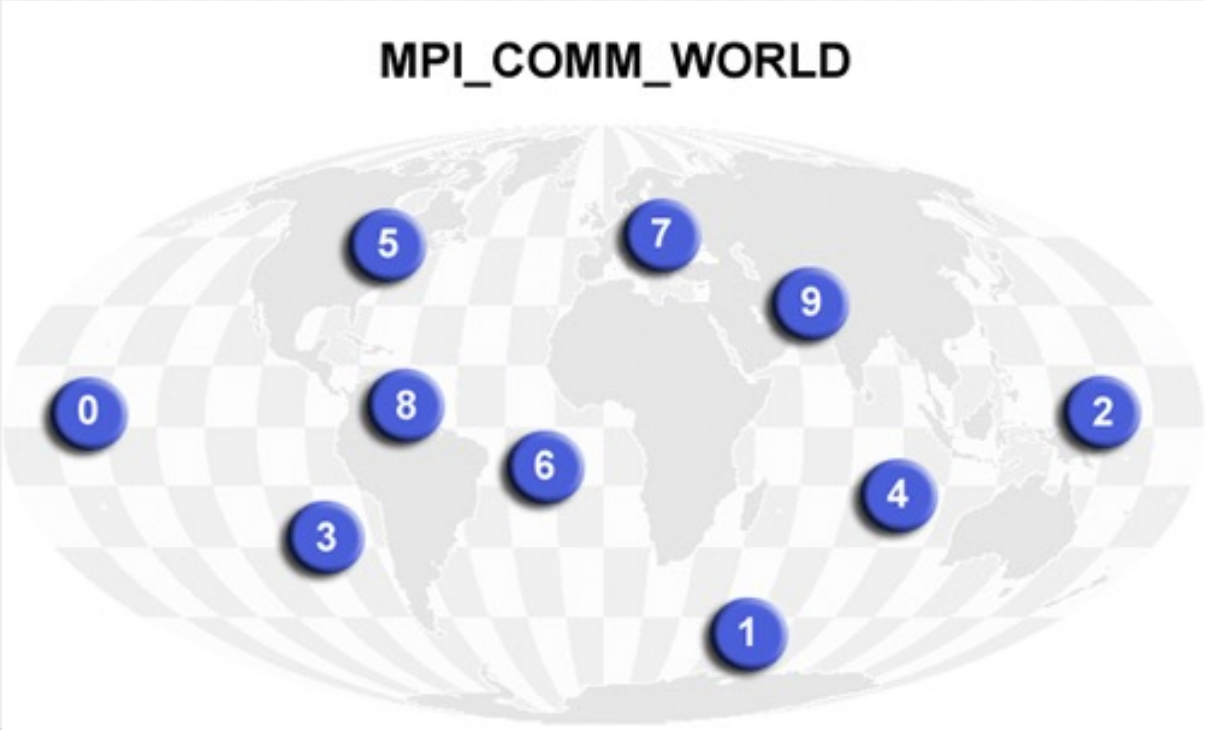
Performance Considerations

- Simple and effective performance model:
  - More parameters == slower
- **contig < vector < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable
- Advice to users:
  - Try datatype “compression” bottom-up

Groups and Communicators

Recap

- Processes belong to **groups**
- Processes within a group are identified with their **rank**
  - A group of n processes has ranks 0 … n-1
- MPI uses objects called **communicators** and groups to define which collection of processes may communicate with each other
  - `MPI_COMM_WORLD` is the default communicator covering all of the original MPI processes



Communicator Basics

- So far we used MPI_COMM_WORLD
  - Allows any process to communicate with any other process
  - Very useful for many tasks
- Sometimes it is advantageous to restrict the number of processes in a communicator (group)
  - E.g. Matrix-Matrix multiplication:
    - Communication along rows and columns
    - Can have individual communicators for rows and columns
  - E.g. Master/Worker:
    - Restrict certain communications only to workers

Groups vs. Communicators

- A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. A group is always associated with a communicator object.

- A communicator encompasses a group of processes that may communicate with each other. All MPI messages must specify a communicator. The communicator that comprises all tasks is MPI_COMM_WORLD.

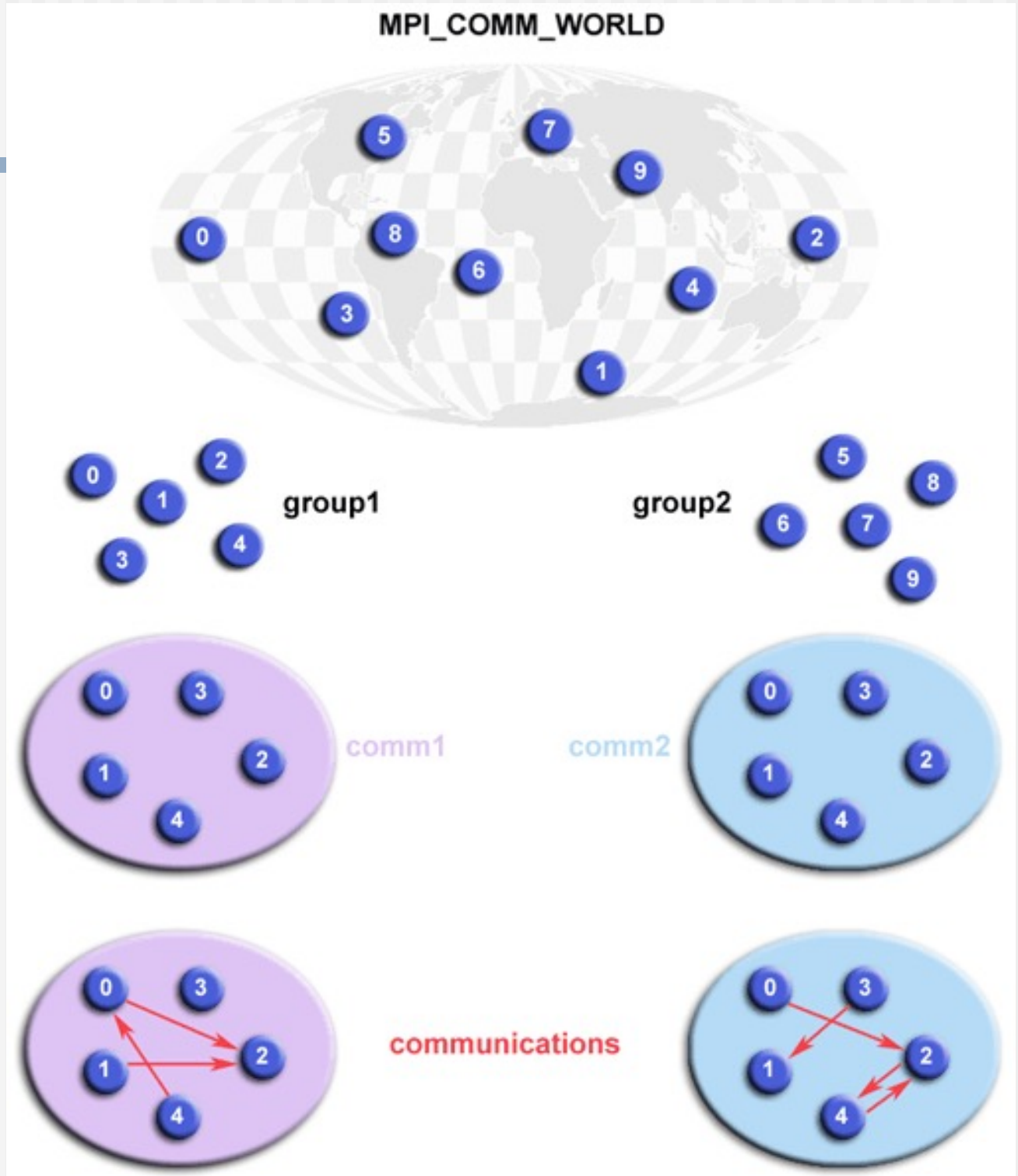
- From the programmer's perspective, a group and a communicator are one. The group routines are primarily used to specify which processes should be used to construct a communicator.

Primary Purposes of Groups and Communicators

1. Allow you to organize tasks, based upon function, into task groups.
2. Enable Collective Communications operations across a subset of related tasks.
3. Provide basis for implementing user defined virtual topologies
4. Provide for safe communications

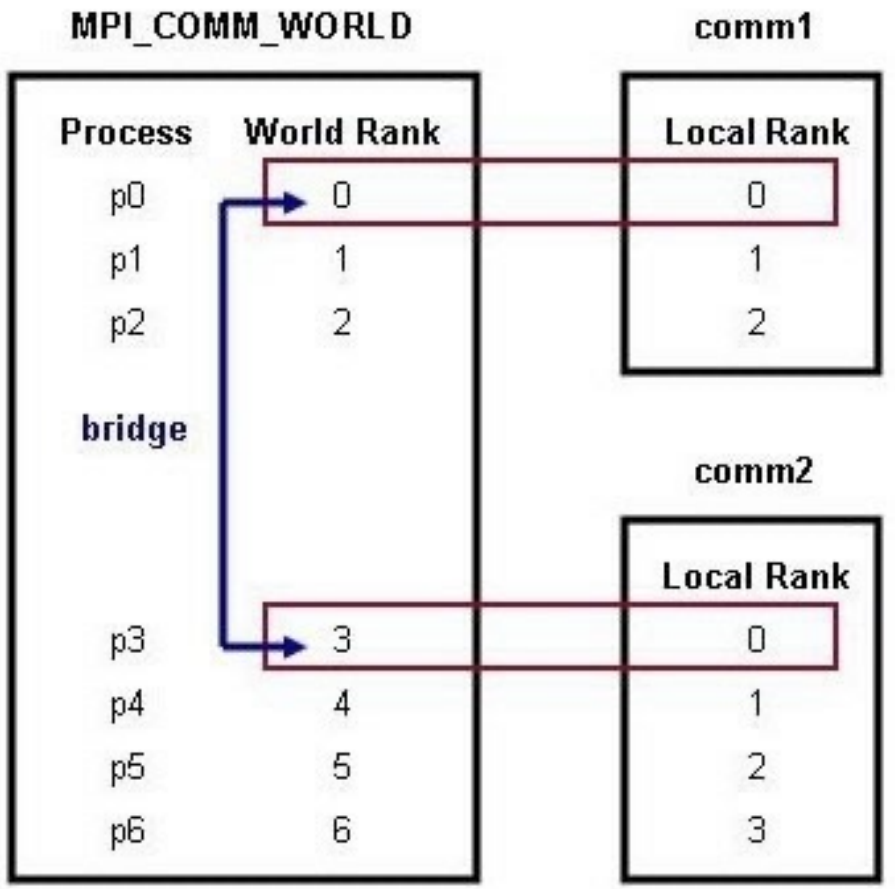
Programming Considerations

- 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.
- 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_WORLD
group1
group2
comm1
comm2
communications

Intra- and Intercommunicators

- Intracommunicators refer to a process group
  - E.g. comm1 from the example below
  - Allow communication within the group
- Intercommunicators refer to two groups of processes
  - Allow communication between disjoint groups



Creation of Intracommunicators

- Split an existing intracommunicator into two or more sub-communicators
- Duplicate an existing intracommunicator
- Modify a group of processes from an existing intracommunicator, and create a new communicator based on this modified group

Communicator Split

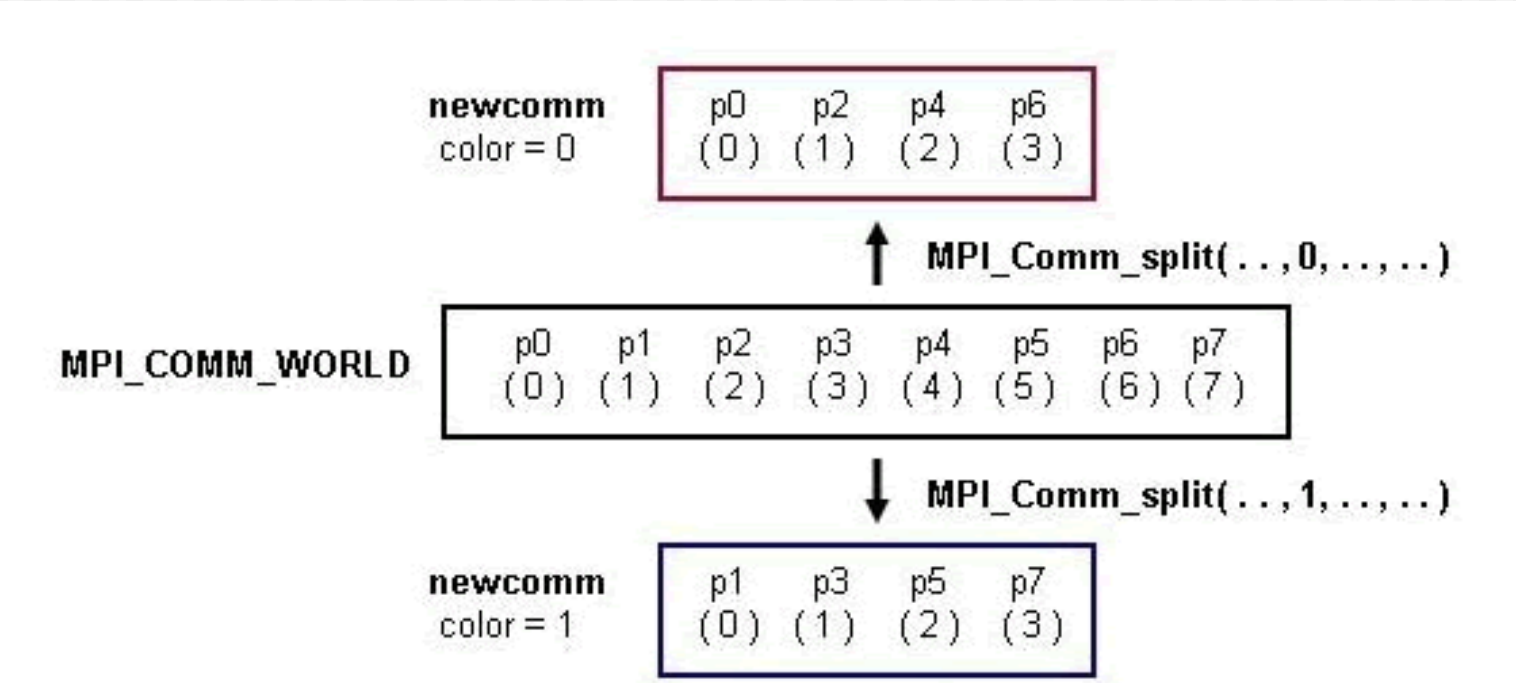
```
MPI_Comm_split(MPI_Comm comm, int color, int key,
               MPI_Comm *newcomm);
MPI_COMM_SPLIT(int comm, int color, int key, int
               newcomm, int IERR)
```

- Color denotes the group a process should be part of
- Key denotes the ranking in the new group

Example

- Split MPI_COMM_WORLD into two groups for even-ranked and odd-ranked process and keep the relative ranking

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
color = rank%2;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &newcomm);
```



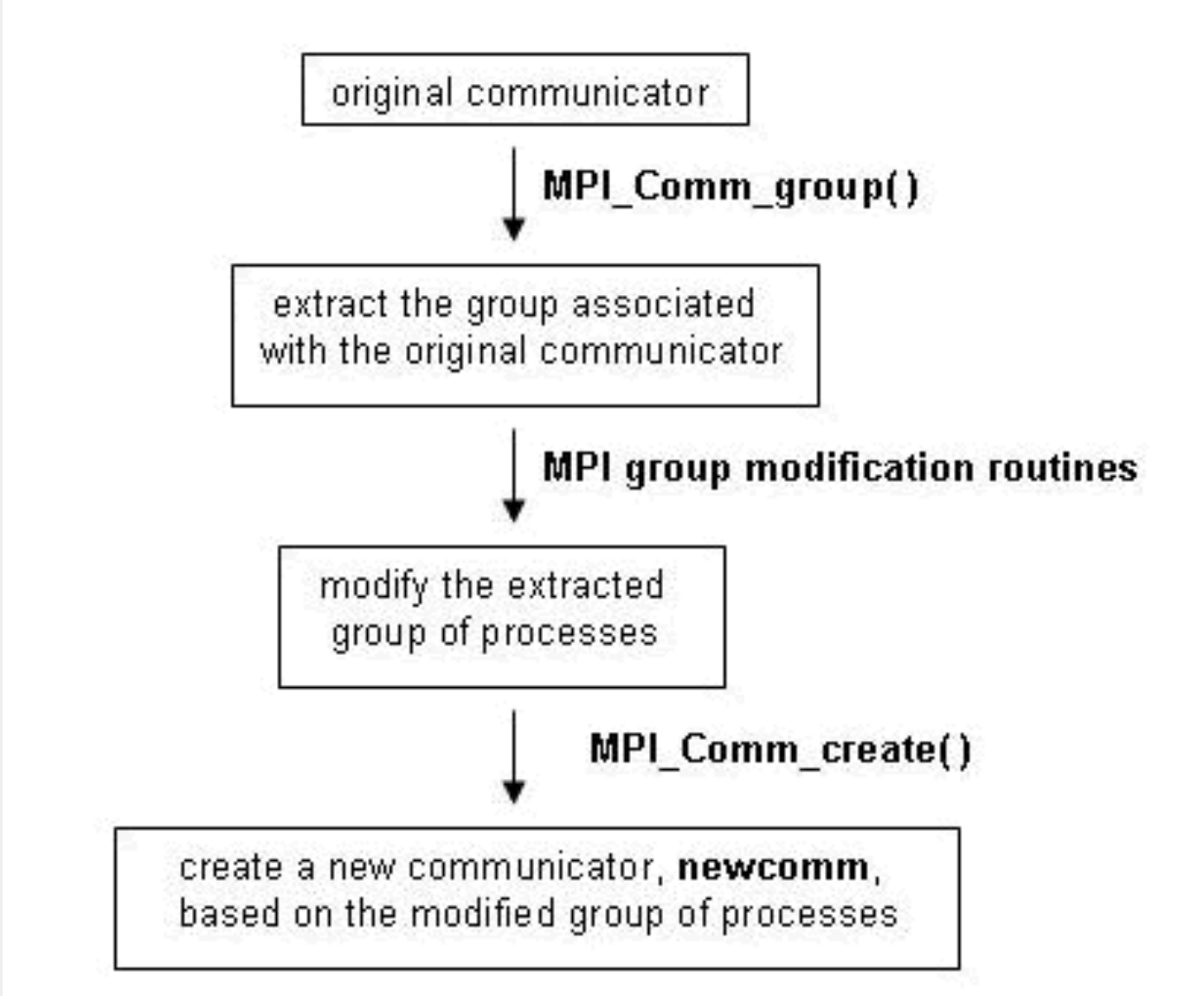
Hands on

- Modify hello-world and create intra-communicators for odd and even processes
- Print out local and global rank

Duplication of existing Communicator

```
MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm);

MPI_COMM_DUP(int comm, int newcomm, int IERR)
```

Modifying a Group of Processes



Group Modifications

- **`MPI_Group_incl`** creates a new group by reordering a specified number of the processes from an existing group
- **`MPI_Group_excl`** creates a new group from an original group that contains all processes left after deleting those with specified ranks.
- **`MPI_Group_union`** creates a new group that contains all processes in the first group followed by all processes in the second group with no duplication of processes.
- **`MPI_Group_intersection`** creates a new group from two groups that contains all processes that are in both of the groups with rank order the same as that in the first group1.
- **`MPI_Group_difference`** creates a new group from two groups that contains all processes in the first group that are not in the second group with rank order the same as that in the first group.

Example

- In a master/worker scheme create communicator for workers
  - Master has rank 0

```
comm_world = MPI_COMM_WORLD;

ranks[0] = 0; /* process 0 not member */

MPI_Comm_group(comm_world, &group_world);

MPI_Group_excl(group_world, 1, &ranks,
               &group_worker);

MPI_Comm_create(comm_world, group_worker, &comm_worker);
…
MPI_Comm_free(&comm_worker);
```

Communicators Summary

- Communicators provide a powerful tool to restrict communication to subsets of processes
- Useful for certain programming styles
  - E.g. Master/Worker
  - Virtual Topologies

Improving Performance

Loss of performance

- Transfer time = latency + message length/bandwidth + synchronization time
- You cannot do much about bandwidth but
- Reduce latency
  - Combine many small into a single large message
  - Hide communication with computation
- Reduce message length
  - Only communicate what is absolutely needed
- Avoid synchronization

Avoid Synchronization

- Synchronization time occurs when
  - Receiver waits for message to be sent
  - Sender waits for message to be received
- Send early, receive late
  - Send early – reduce time receiver has to wait for message
  - Receive late – do as much work as possible on the receiving side before waiting for message to arrive
- BUT: What if underlying protocol requires send/receive handshake? Then things are actually getting worse!

Avoid Synchronization

- Non-blocking communication modes can help
  - Post Irecv early on so that send would find matching receive
  - But could introduce buffer problems

- If receiving order is not important avoid receiving from a dedicated sender but post receives with `MPI_ANY_SOURCE`

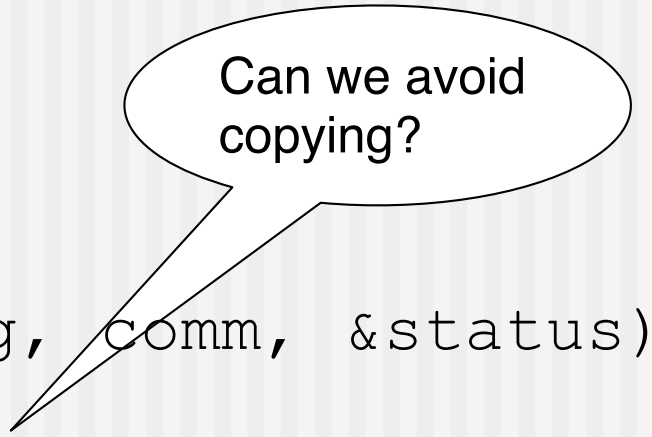
```
MPI_Recv(buffer, size, MPI_INT,
         MPI_ANY_SOURCE, tag, comm, &status)
```

MPI-ANY-SOURCE Example

```
if (myrank == 0) {
    for (int i = 1,numproc-1) {
        MPI_Recv(b[i], size, MPI_INT, i, tag,
                 comm, &status);
    }
} else {
  MPI_Send(x, size, MPI_INT, 0, tag, comm);
}
```

- Better:

```
MPI_Recv(x, size, MPI_INT,
         MPI_ANY_SOURCE, tag, comm, &status);
b[status.MPI_SOURCE] = x;
```



Example Cont'd

```
MPI_Probe(MPI_ANY_SOURCE, tag, comm, &status);

MPI_Recv(b[status.MPI_SOURCE], size, MPI_INT,
         status.MPI_SOURCE, tag, comm,
         &status);
```

Avoid Synchronization

- Use Sendrecv
- Use Collective operations
  - Most of them will synchronize but are typically implemented well.
  - But avoid MPI_Barrier and all-to-all
- Pitfall:
  - Not all MPI implementations are equally well optimized
  - If critical, implement several variants and compare their timing (same for derived datatypes)

Latency Hiding

- Use non-blocking communication and try to do as much computation as possible before blocking on the WAIT
  - Use standard send/receive if WAIT follows immediately after the send/receive
  - Can result in buffer and/or envelope queue overflow

Reduce communication

- Re-compute vs. communication
  - Sometimes it can be more efficient to compute certain data on all processes where it is needed rather than communicating it.

Summary

- Several ways to reduce communication/synchronization overhead
- Use tools to figure out where the hot-spots of your application are
- Most performance tuning is NOT portable and highly implementation and hardware dependent