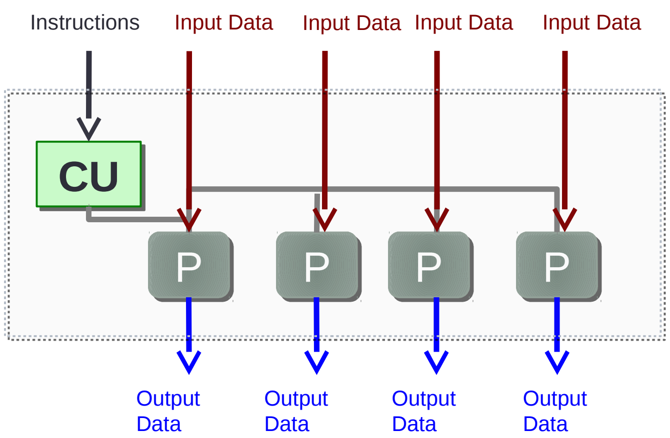
DISTRIBUTED MEMORY PROGRAMMING WITH MPI

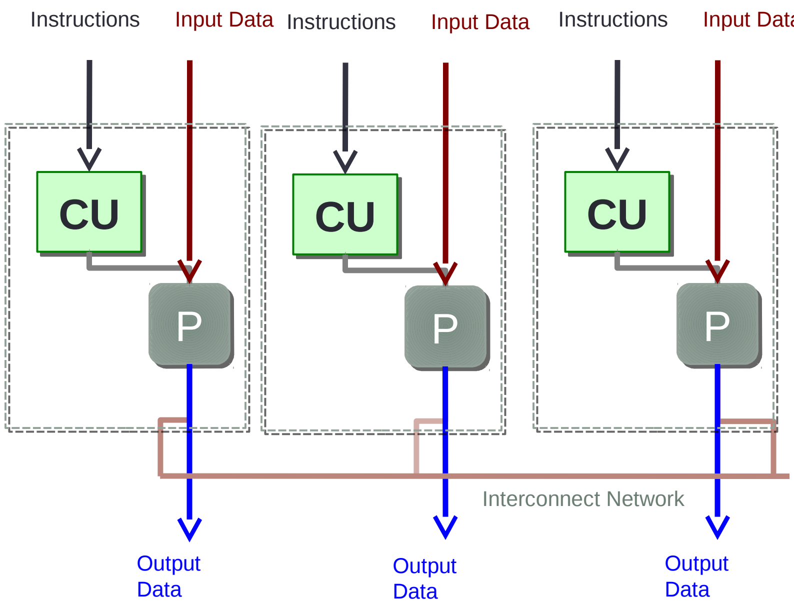
Carlos Jaime Barrios Hernández, PhD.

Remember Special Features of Architecture

- Remember “concurrency”: it exploits better the resources (shared) within a computer.
- Exploit SIMD and MIMD Architectures

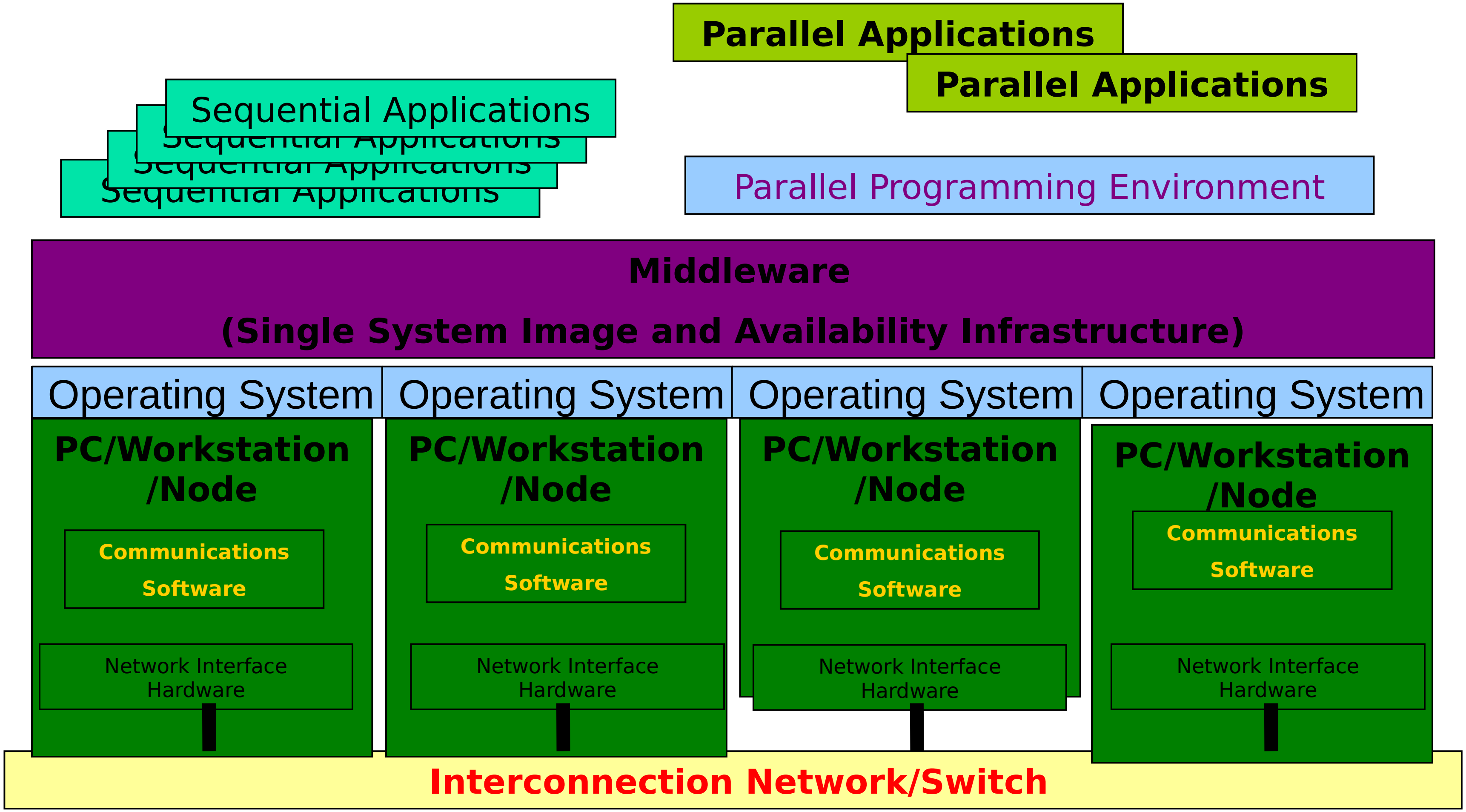


SIMD



MIMD

Cluster Computing Architecture



Distributed Computing Paradigms

- Communication Models:
  - Message Passing
  - Shared Memory
- Computation Models:
  - Functional Parallel
  - Data Parallel

Message Passing

- A process is a program counter and address space.
- Message passing is used for communication among processes.
- Inter-process communication:
  - Type:

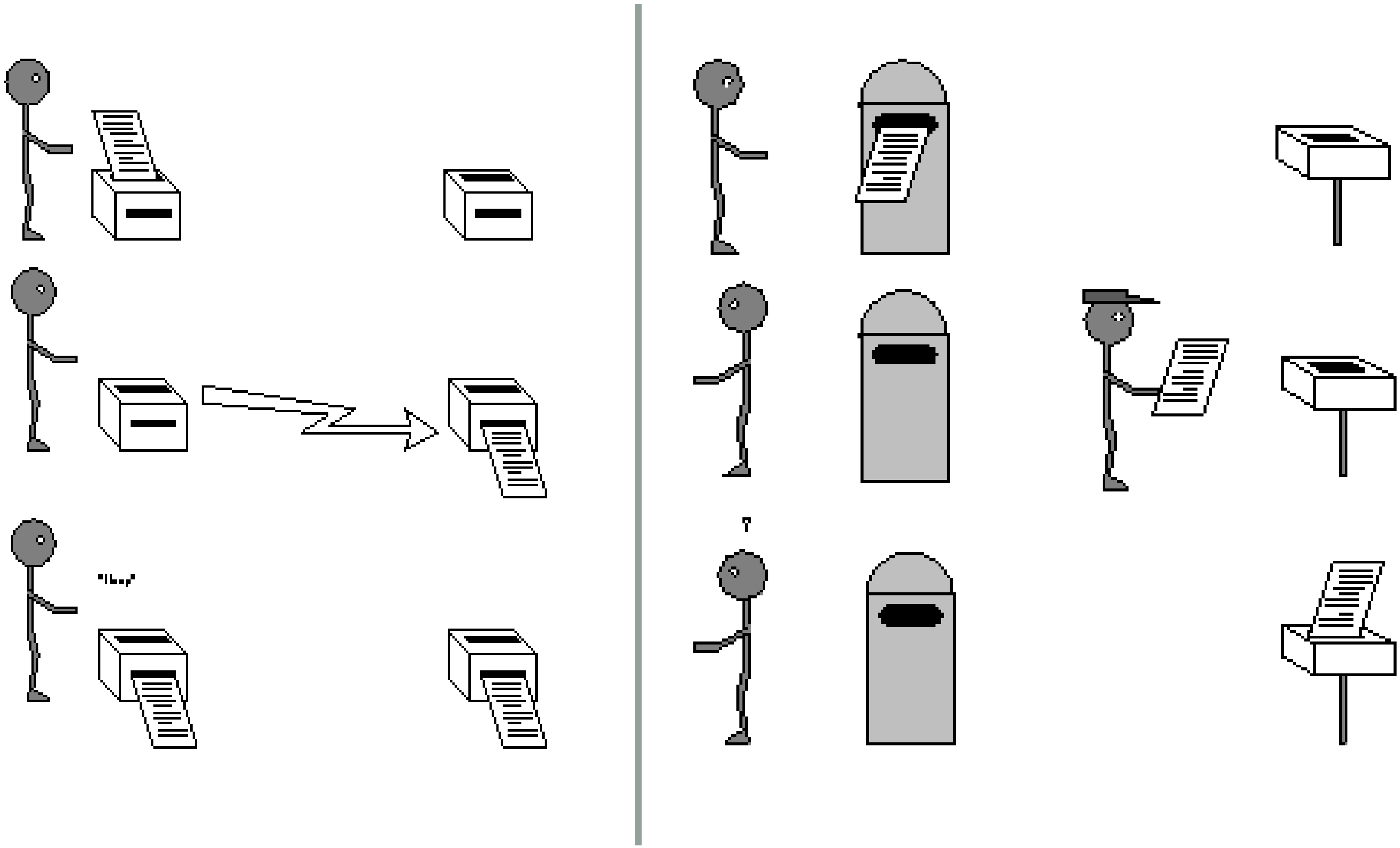
    Synchronous / Asynchronous
  - Movement of data from one process's address space to another's

Synchronous Vs. Asynchronous

- A synchronous communication is not complete until the message has been received.

- An asynchronous communication completes as soon as the message is on the way.

Synchronous Vs. Asynchronous (cont.)



What is message passing?

- Data transfer.
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

- **MPI in a nutshell**
  - It is a library specification
  - Works natively with C and Fortran
  - Not a specific implementation or product
  - Scalable
    - Must handle multiple machines
  - Portable
    - Sockets API change from one OS to another
    - Handles Big-endian/little-endian architectures
  - Efficient
    - Optimized communication algorithms
    - Allow communication and computation overlap

MPI – Message Passing Interface

MPI – Message Passing Interface

- **MPI References**
  - Books
    - Using MPI: Portable Parallel Programming with the Message Passing Interface, by Gropp, Lusk, and Skejellum, MIT Press, 1994.
    - MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
    - Parallel Programming with MPI, by Peter Pacheco, Morgan Kaufmann, 1997.
  - The standard:
    - at http://www.mpi-forum.org

MPI History

- 1990 PVM: Parallel Virtual Machine (Oak Ridge Nat'l Lab)
  - Message-passing routines
  - Execution environment (spawn + control parallel processes)
  - No an industry standard
- 1992 meetings (Workshop, Supercomputing'92)
- 1993 MPI draft
- 1994 MPI Forum (debates)
- 1994 MPI-1.0 release (C & Fortran bindings) + standardization
- 1995 MPI-1.1 release
- 1997 MPI-1.2 release (errata) + MPI-2 release (new features, C++ & Fortran 90 bindings)
- ???? MPI-3 release (new: FT, hybrid, p2p, RMA, …)
- 2000 MPI (ch), Madeline, V4….
- 2005 OpenMPI…

MPI Programming

- **MPI**
  - Use of a single program, on multiple data
  - What does it do?
    - way of identifying process
    - Independent of low-level API
    - Optimized communication
    - Allow communication and computation overlap
  - What does it do not?
    - gain performance of application for free
    - application must be adapted

Features of MPI

- General
  - Communications combine context and group for message security.
  - Thread safety can’t be assumed for MPI programs.

Features that are NOT part of MPI

- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory

Why to use MPI?

- MPI provides a powerful, efficient, and portable way to express parallel programs.

- MPI was explicitly designed to enable libraries which may eliminate the need for many users to learn (much of) MPI.

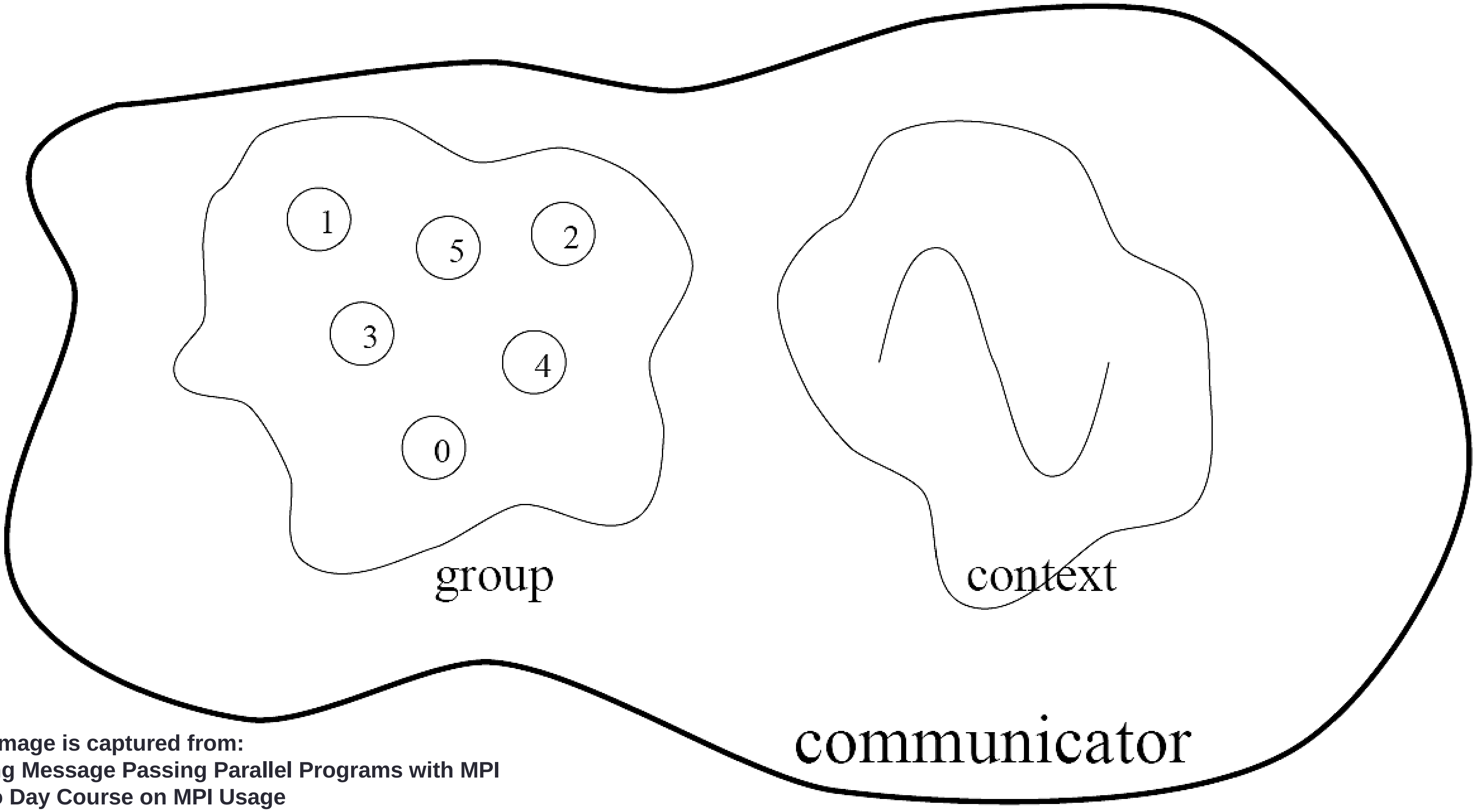
- Portable !!!!!!!!!!!!!!!!!!!!!!!!!!

- Good way to learn about subtle issues in parallel computing

How big is the MPI library?

- Huge (125 Functions).
- Basic (6 Functions).

Group and Context



This image is captured from:
Writing Message Passing Parallel Programs with MPI
A Two Day Course on MPI Usage
Course Notes
Edinburgh Parallel Computing Centre
The University of Edinburgh

Group and Context (cont.)

- Are two important and indivisible concepts of MPI.
- Group: is the set of processes that communicate with one another.
- Context: it is somehow similar to the frequency in radio communications.
- Communicator: is the central object for communication in MPI. Each communicator is associated with a group and a context.

Communication Modes

- Based on the type of send:
  - Synchronous: Completes once the acknowledgement is received by the sender.
  - Buffered send: completes immediately, unless if an error occurs.
  - Standard send: completes once the message has been sent, which may or may not imply that the message has arrived at its destination.
  - Ready send: completes immediately, if the receiver is ready for the message it will get it, otherwise the message is dropped silently.

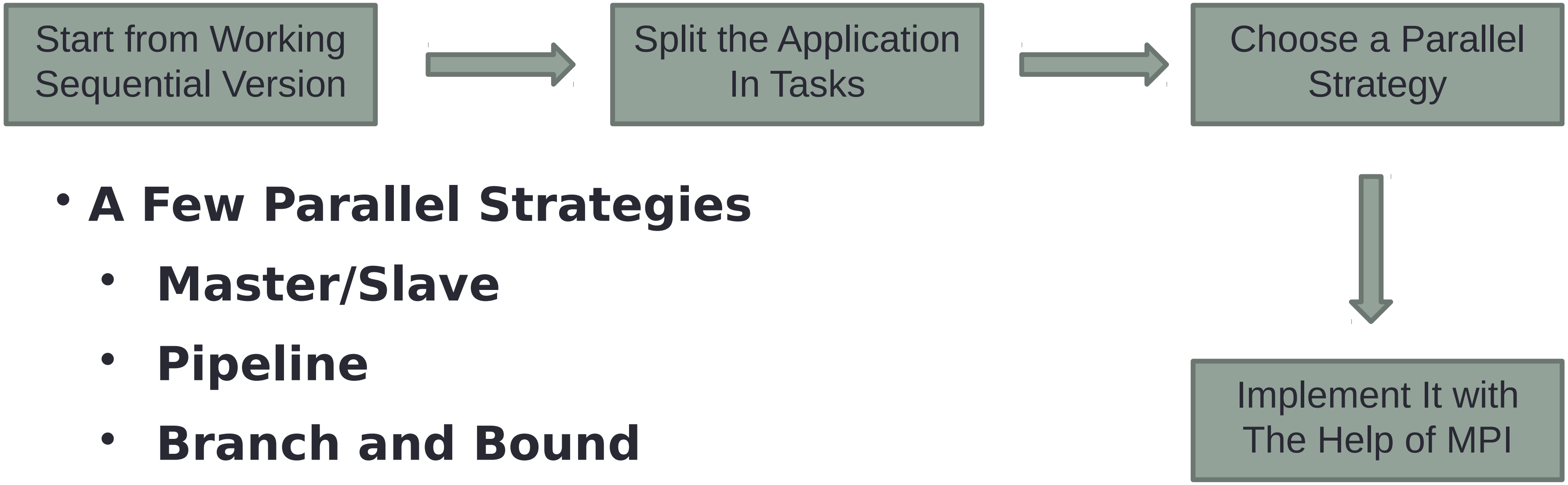
Blocking vs. Non-Blocking

- Blocking, means the program will not continue until the communication is completed.

- Non-Blocking, means the program will continue, without waiting for the communication to be completed.

MPI Programming

- **Possible Programming Workflow**



- **A Few Parallel Strategies**
  - **Master/Slave**
  - **Pipeline**
  - **Branch and Bound**

Parallel Strategies

- **Master/Slave**
  - **Master is one process that centrilizes all tasks**
  - **Slaves starve for work**

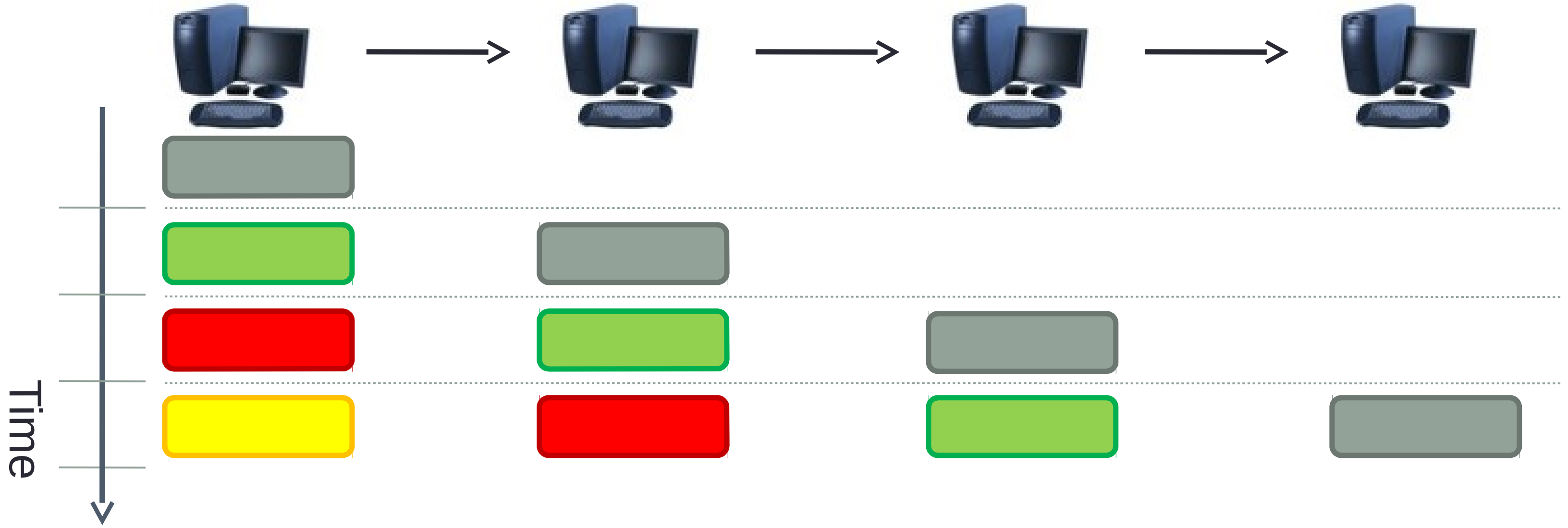
Parallel Strategies

- **Master/Slave**
  - **Master is often the bottleneck**
  - **Scalability is limited due to centralization**
  - **Possible to use replication to improve performance**
  - **It is adatable to heterogenous platforms**

Parallel Strategies

- **Pipeline**
  - **Each process plays a specific role, pipeline stages**
  - **Data follows in a single direction**
  - **Parallelism is achieved when the pipeline is full**

Task 1
Task 2
Task 3
Task 4

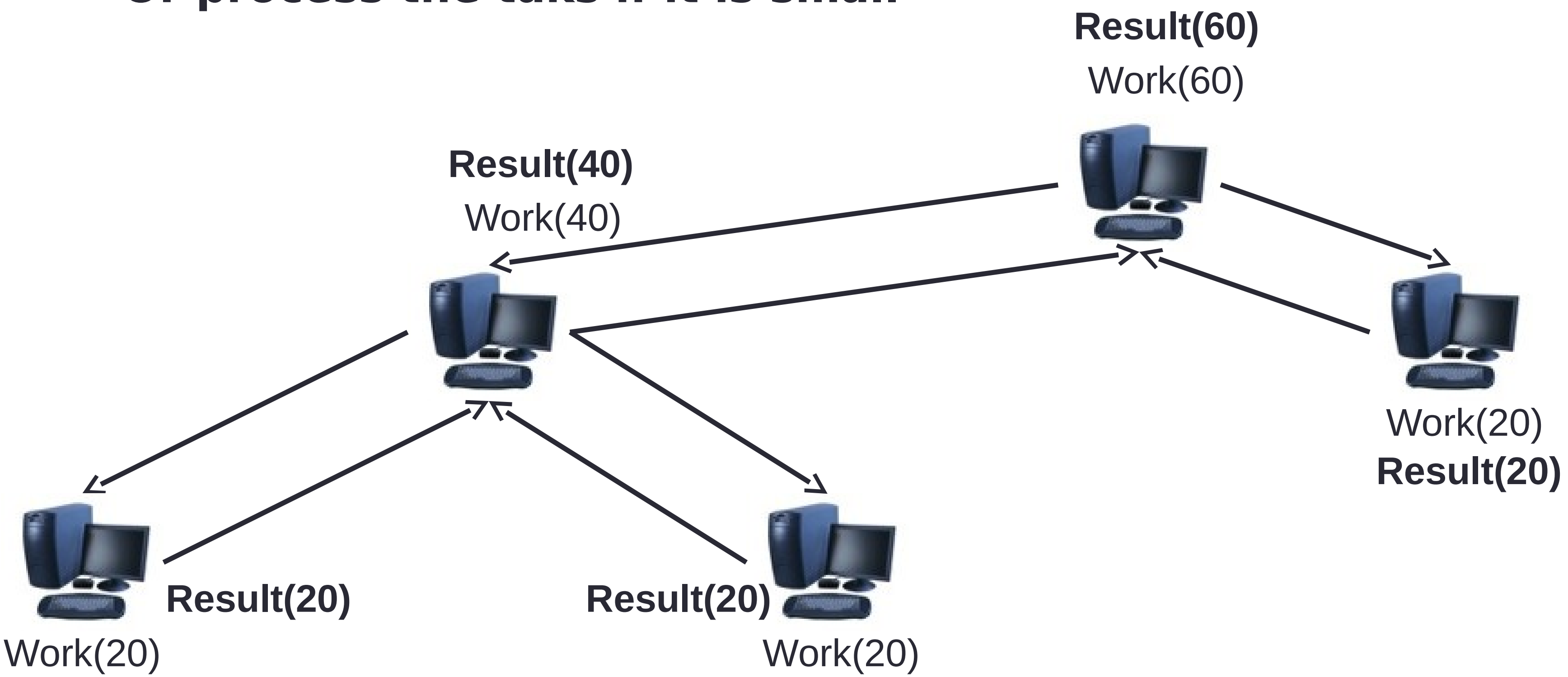


Parallel Strategies

- **Pipeline**
  - **Scalabillity is limited by the number of stages**
  - **Synchronization may lead to bubbles**
    - **Slow sender**
    - **Fast receiver**
  - **Difficult to use on heterogenous platforms**

Parallel Strategies

- Divide and Conquer
  - Recursevely partion task on roughly equal sized tasks
  - Or process the taks if it is small



Parallel Strategies

- **Divide and Conquer**
  - **More scalable**
  - **Possible to use replicated branches**
  - **In practice is difficult to split tasks**
  - **Suitable for branch and bound algorithms**

MPI Programming

- **Installing**
  - Some common MPI implementations, all free:
    - OpenMPI
      http://www.open-mpi.org/
    - MPICH-2
      http://www.mcs.anl.gov/research/projects/mpich2/
    - LAM/MPI
      http://www.lam-mpi.org/

MPI Programming

- **Installing**
  - I’m using MPICH-2
  - Installed in Ubuntu 10.04 Lucid Lynx with

```
$ sudo apt-get install mpich2
```

  - Should work for most Debian based distributions
  - Must create a local configuration file

```
$ echo “MPD_SECRET_WORD=ChangeMe” > ~/.mpd.conf
```

- **Test program**

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

int main(int argc, char **argv){

/* Initialize MPI */
MPI_Init(&argc, &argv);

printf("Test Program\n");

/* Finalize MPI */
return MPI_Finalize();
}
```

Skeleton MPI Program

```
#include <mpi.h>

main( int argc, char** argv )
{
    MPI_Init( &argc, &argv );

    /* main part of the program */

  /*
    Use MPI function call depend on your data
  partitioning and the parallelization
  architecture
  */

    MPI_Finalize();
}
```

A minimal MPI program(c)

```
#include “mpi.h”
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf(“Hello, world!\n”);
  MPI_Finalize();
  Return 0;
}
```

- **Compiling**
  - Compiled with gcc, but a mpicc script is provided to invoke gcc with specific MPI options enabled

```
$ mpicc mpi_program.c –o my_mpi_executable
```

  - Executed with a specital script

```
$ mpirun –np 1 my_mpi_executable
$ mpirun –np 2 my_mpi_executable
$ mpirun –np 3 my_mpi_executable
```

MPI Programming

- **Running**
  - Compiled with gcc, but a mpicc script is provided to invoke gcc with specific mpi functions

```
$ mpicc mpi_program.c –o my_mpi_executable
```

  - For a complete list of parameters try

```
$ man mpicc
```

  - Executed with a specital scrip

```
$ mpirun –np 2 my_mpi_executable
```

A minimal MPI program(c) (cont.)

- #include “mpi.h” provides basic MPI definitions and types.
- MPI_Init starts MPI
- MPI_Finalize exits MPI
- Note that all non-MPI routines are local; thus “printf” run on each process
- Note: MPI functions return error codes or MPI_SUCCESS

Error handling

- By default, an error causes all processes to abort.
- The user can have his/her own error handling routines.
- Some custom error handlers are available for downloading from the net.

Improved Hello (c)

```
>include <mpi.h#
#include <stdio.h>
int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("I am %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

MPI Programming

- **How many processing units are available?**

```
int MPI_Comm_size(MPI_Comm comm, int *psize)
```

  - Group of process to communicate
  - Default Communicator: For grouping all process use **MPI_COMM_WORLD**
- **psize**
  - Passed as reference will return the total amoung of proccess in this communicator

Data Types

- The data message which is sent or received is described by a triple (address, count, datatype).
- The following data types are supported by MPI:
  - Predefined data types that are corresponding to data types from the programming language.
  - Arrays.
  - Sub blocks of a matrix
  - User defined data structure.
  - A set of predefined data types

Basic MPI types

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_SHORT	signed short
MPI_UNSIGNED_SHORT	unsigned short
MPI_INT	signed int
MPI_UNSIGNED	unsigned int
MPI_LONG	signed long
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

MPI Programming

- **Exercise 1 – Hello World**
  - **Create program that prints hello world and the total number of available process on the screen**
  - **Use –np with a variable number to verify that your program is working**

MPI Programming

- **Exercise 2 – Who am I**
  - **If I am process 0**
    - **Prints: “hello world”**
  - **else**
    - **Prints: “I’m process <ID>”**
    - **Replacing <ID> by the process rank**

Why defining the data types during the send of a message?

Because communications take place between heterogeneous machines. Which may have different data representation and length in the memory.

MPI blocking send

```
MPI_SEND(void *start, int
 count,MPI_DATATYPE datatype, int dest,
 int tag, MPI_COMM comm)
```

- The message buffer is described by (start, count, datatype).
- dest is the rank of the target process in the defined communicator.
- tag is the message identification number.

MPI blocking receive

```
 MPI_RECV(void *start, int count,
MPI_DATATYPE datatype, int source, int tag,
MPI_COMM comm, MPI_STATUS *status)
```

- Source is the rank of the sender in the communicator.
- The receiver can specify a wildcard value for souce (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable
- Status is used for exrtra information about the received message if a wildcard receive mode is used.
- If the count of the message received is less than or equal to that described by the MPI receive command, then the message is successfully received. Else it is considered as a buffer overflow error.

MPI_STATUS

- Status is a data structure
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(…, MPI_ANY_SOURCE, MPI_ANY_TAG, …,
 &status)
recvd_tag = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);
```

More info

- A receive operation may accept messages from an arbitrary sender, but a send operation must specify a unique receiver.

- Source equals destination is allowed, that is, a process can send a message to itself.

Why MPI is simple?

- Many parallel programs can be written using just these six functions, only two of which are non-trivial;
  - MPI_INIT
  - MPI_FINALIZE
  - MPI_COMM_SIZE
  - MPI_COMM_RANK
  - MPI_SEND
  - MPI_RECV

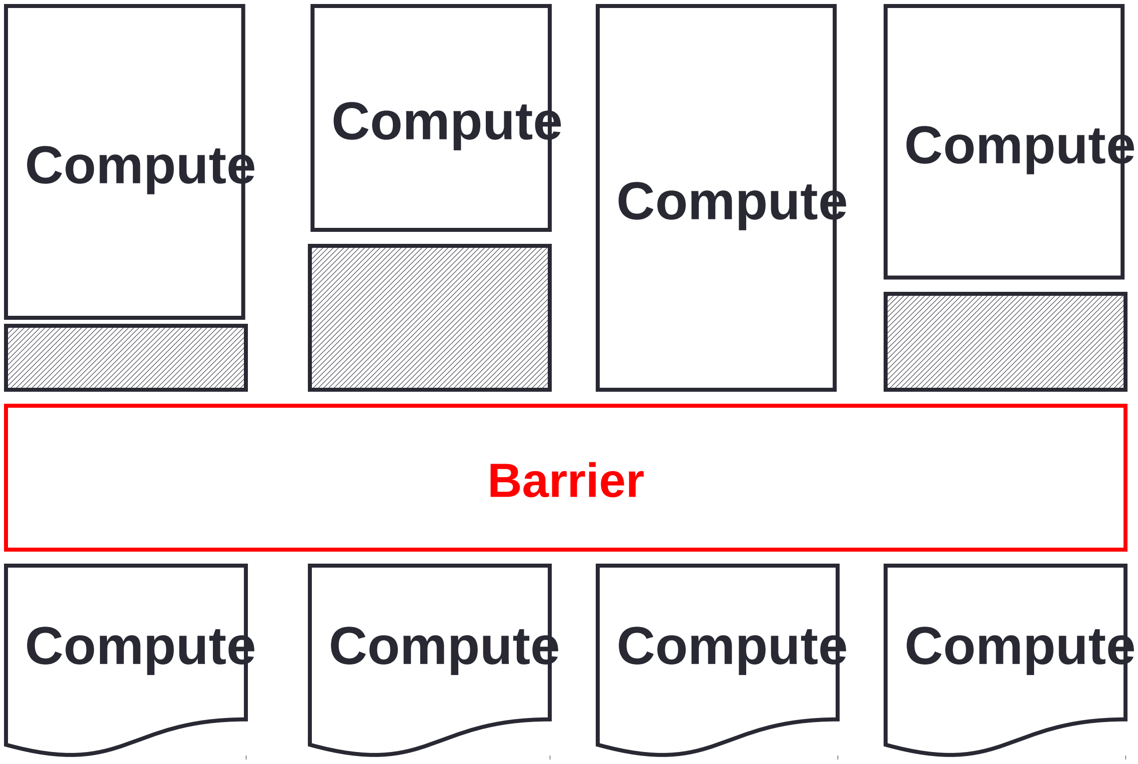
Collective Communications

- Point-to-point communications involve pairs of processes.
- Many message passing systems provide operations which allow larger numbers of processes to participate

Types of Collective Transfers

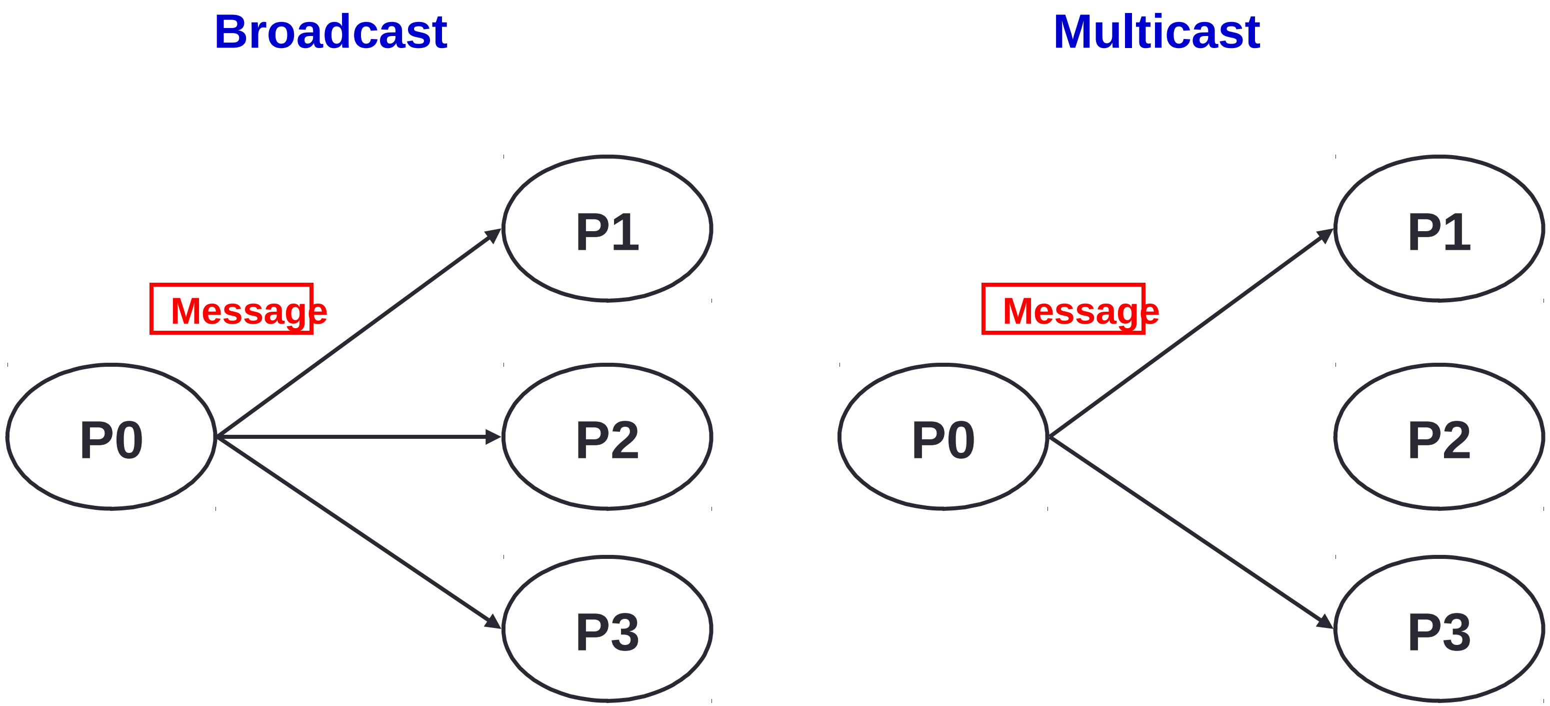
- Barrier
  - Synchronizes processors
  - No data is exchanged but the barrier blocks until all processes have called the barrier routine
- Broadcast (sometimes multicast)
  - A broadcast is a one-to-many communication
  - One processor sends one message to several destinations
- Reduction
  - Often useful in a many-to-one communication

Barrier

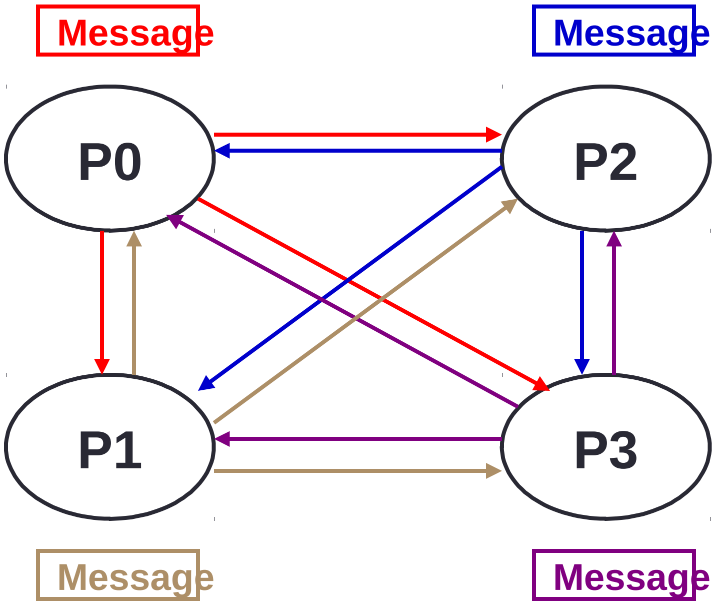


Compute
Compute
Compute
Compute
Barrier
Compute
Compute
Compute
Compute

Broadcast and Multicast

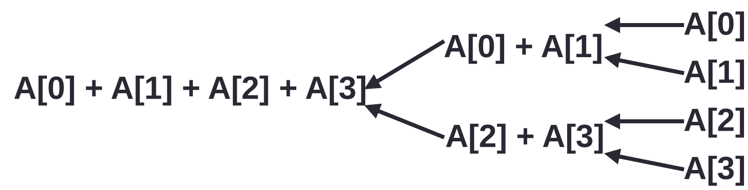


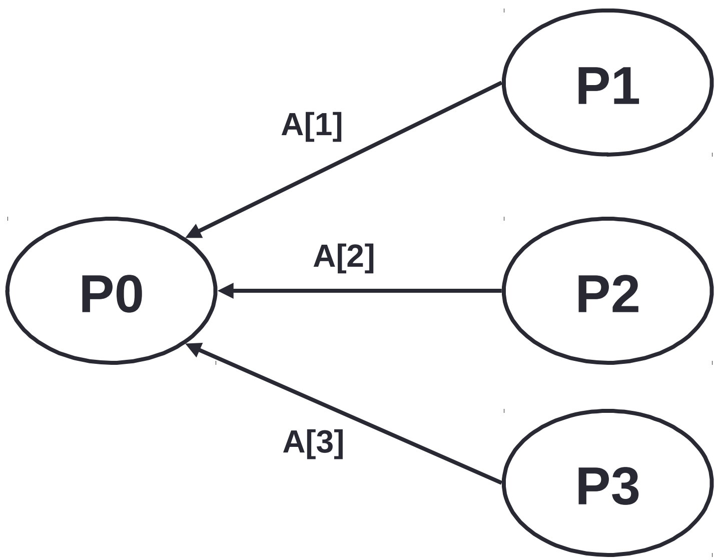
All-to-All

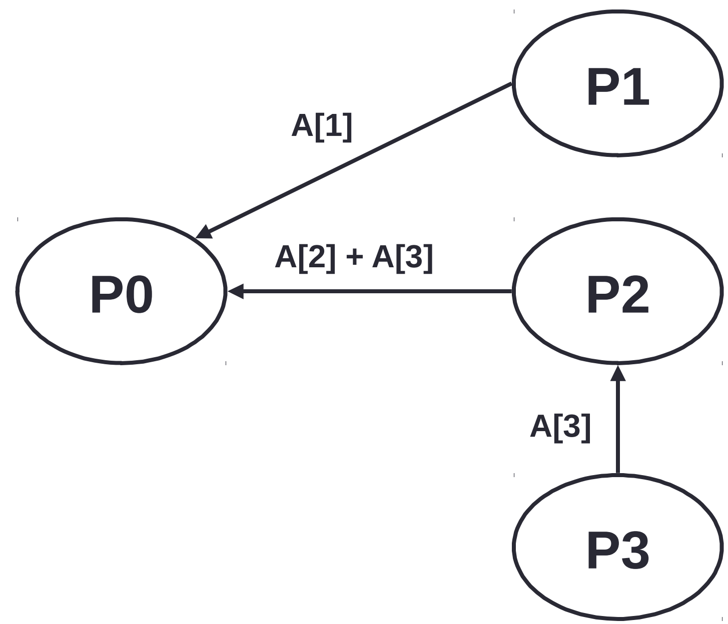


Reduction

```
sum ← 0
for i ← 1 to p do
    sum ← sum + A[i]
```





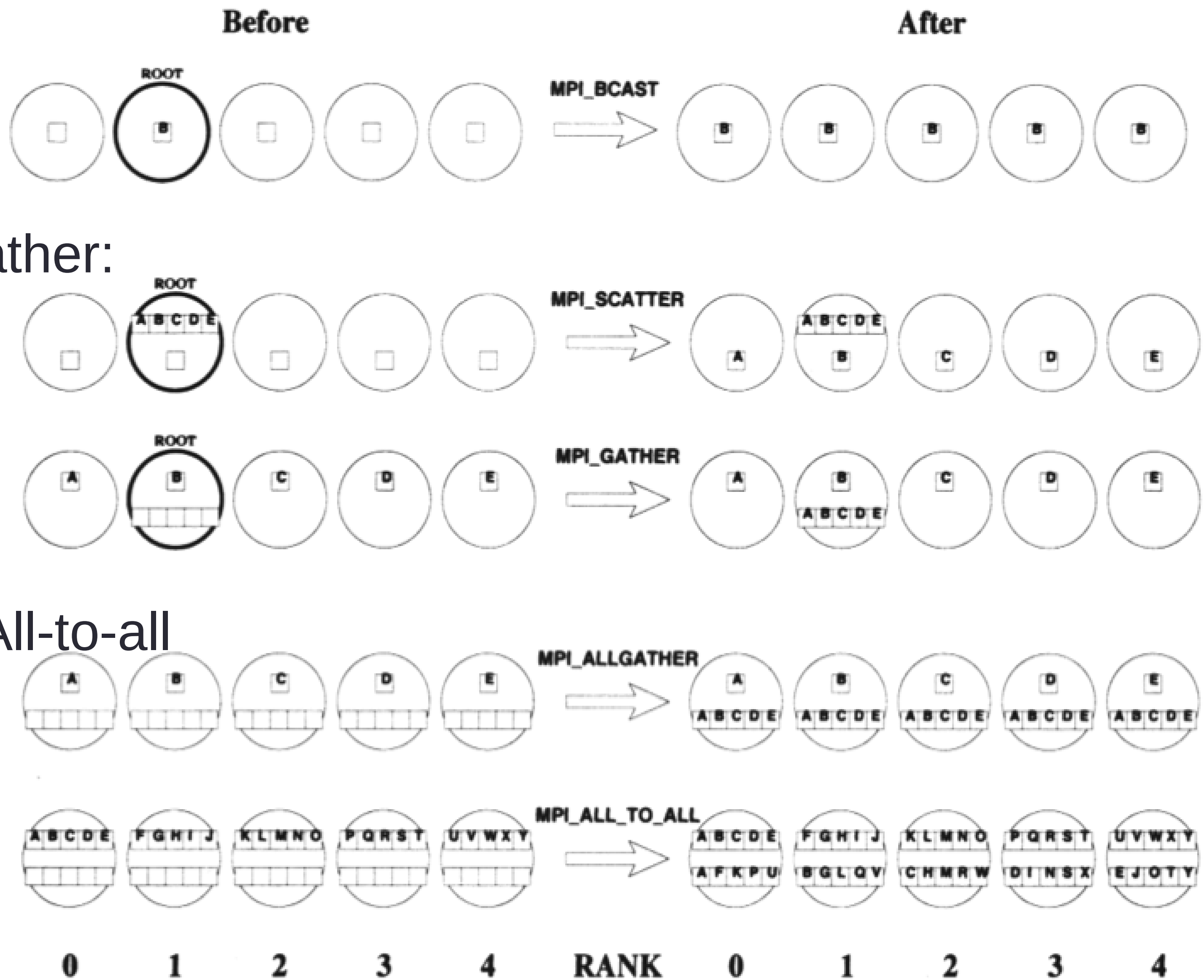


Introduction to Collective Operations in MPI

- Collective ops are called by all processes in a communicator.
  - No tags
  - Blocking
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE/ALLREDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.
- Others:
  - **MPI_[ALL]SCATTER[V]/[ALL]GATHER[V]**

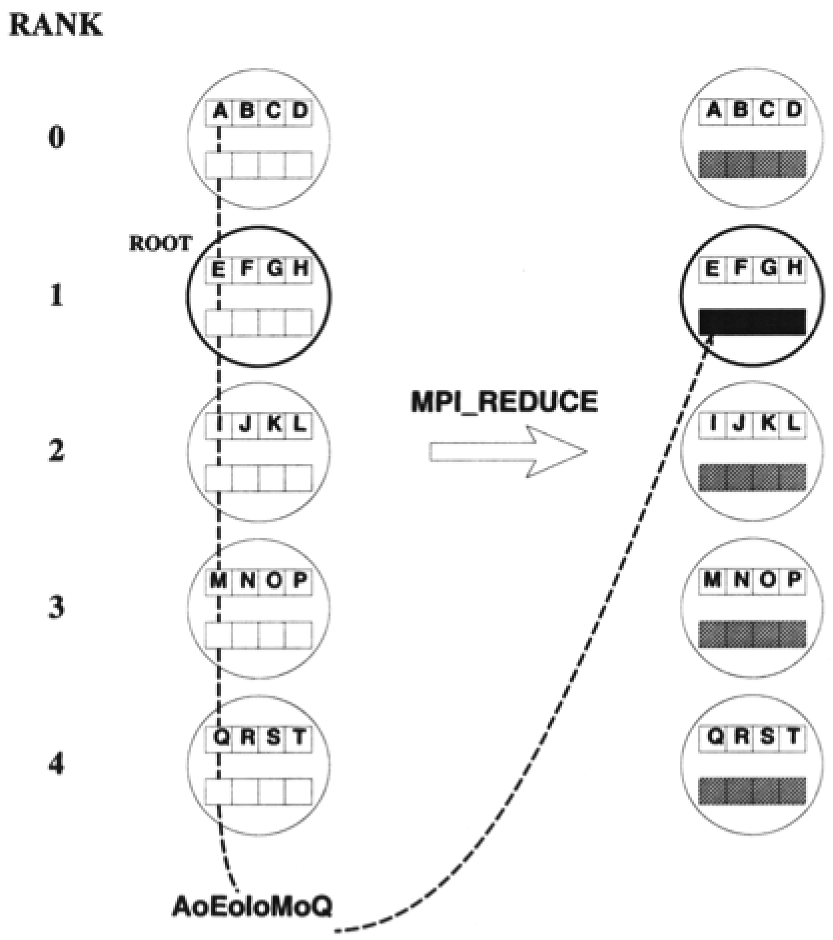
Collectives at Work

- BCAST:
- Scatter/Gather:
- Allgather/All-to-all



Collectives at Work (2)

- Reduce:

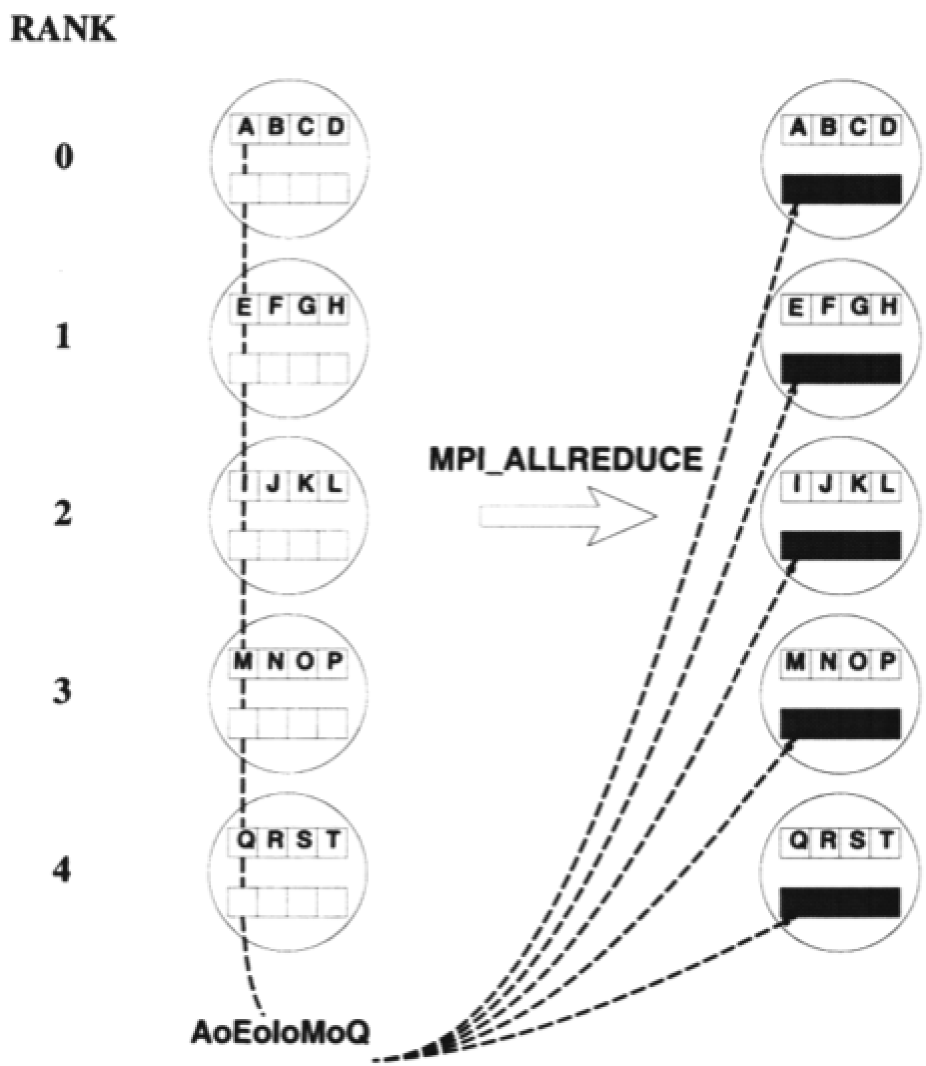


- Predefined Ops (assocociative & commutative) / user ops (assoc.)

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

Collectives at Work (3)

- Allreduce:



Simple full example

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

int main(int argc, char *argv[])
{
  const int tag = 42;        /* Message tag */
  int id, ntasks, source_id, dest_id, err, i;
  MPI_Status status;
  int msg[2]; /* Message array */

  err = MPI_Init(&argc, &argv); /* Initialize MPI */
  if (err != MPI_SUCCESS) {
    printf("MPI initialization failed!\n");
    exit(1);
  }
  err = MPI_Comm_size(MPI_COMM_WORLD, &ntasks); /* Get nr of tasks */
  err = MPI_Comm_rank(MPI_COMM_WORLD, &id);  /* Get id of this process */
  if (ntasks < 2) {
    printf("You have to use at least 2 processors to run this program\n");
    MPI_Finalize();   /* Quit if there is only one processor */
    exit(0);
  }
```

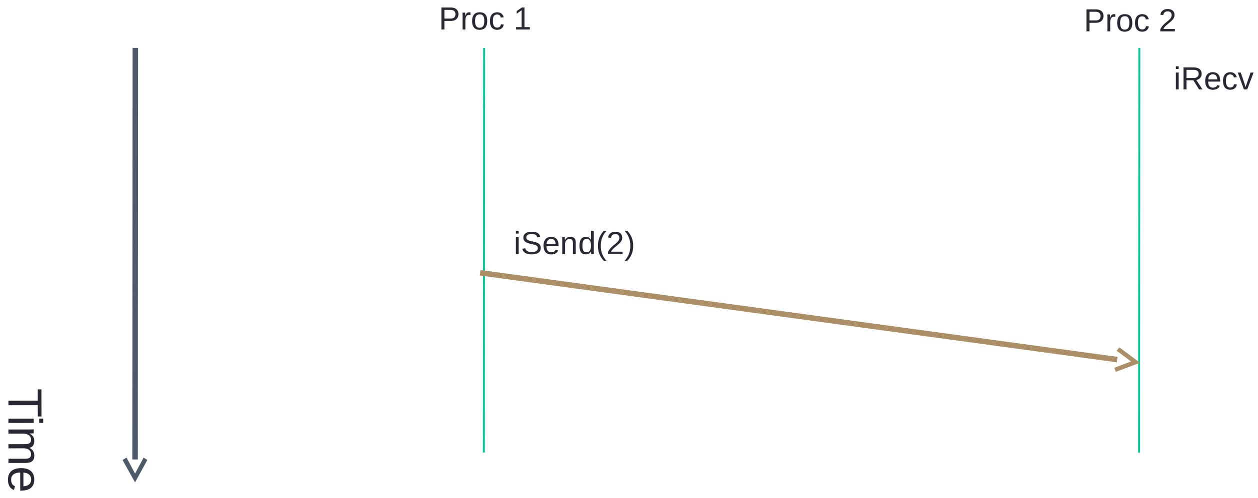
Simple full example (Cont.)

```
if (id == 0) {  /* Process 0 (the receiver) does this */
    for (i=1; i<ntasks; i++) {
      err = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, \
                     &status);          /* Receive a message */
      source_id = status.MPI_SOURCE; /* Get id of sender */
      printf("Received message %d %d from process %d\n", msg[0], msg[1], \
             source_id);
    }
  }
  else {    /* Processes 1 to N-1 (the senders) do this */
    msg[0] = id; /* Put own identifier in the message */
    msg[1] = ntasks;        /* and total number of processes */
    dest_id = 0; /* Destination address */
    err = MPI_Send(msg, 2, MPI_INT, dest_id, tag, MPI_COMM_WORLD);
  }

  err = MPI_Finalize();         /* Terminate MPI */
  if (id==0) printf("Ready\n");
  exit(0);
  return 0;
}
```

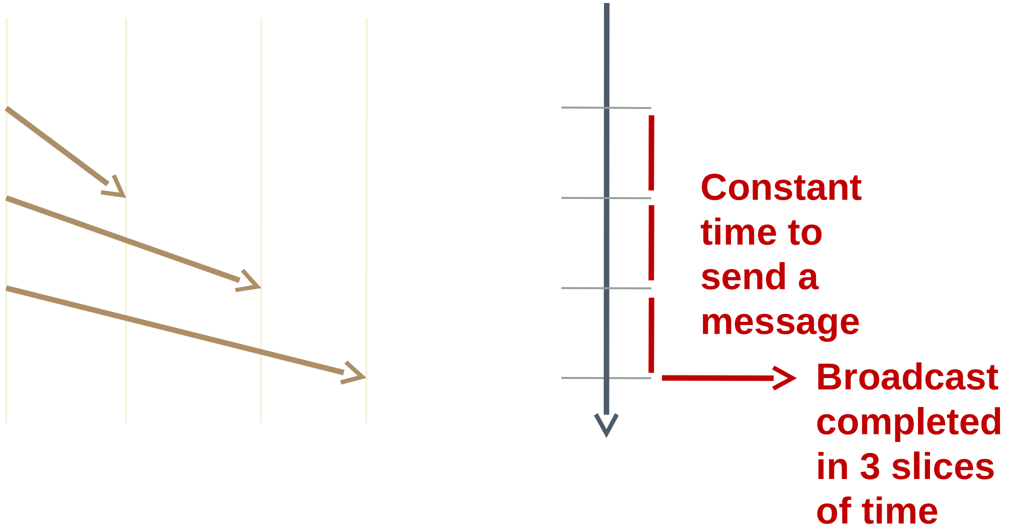
MPI
One-to-one Communication

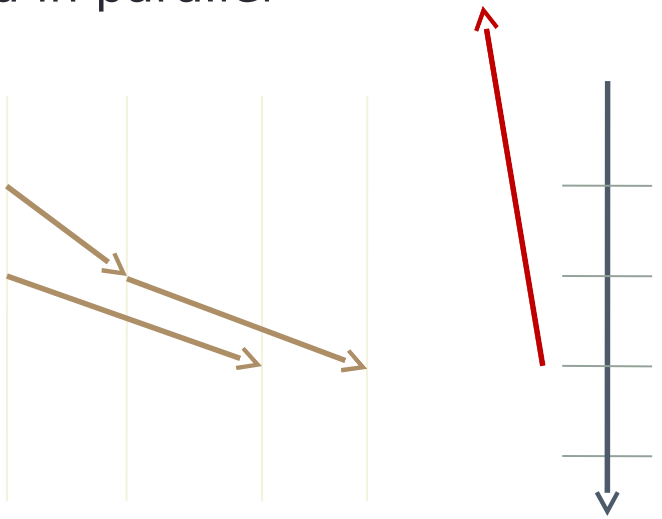
- **Assynchronous/Non-Blocking**
  - Process signs it is waiting for a message
  - Continue working meanwhile



MPI
Collective Communication

- **Proccess master wants to send a message to everybody**
  - First solution, process master send N-1 messages
  - Optimized collective communication send in parallel

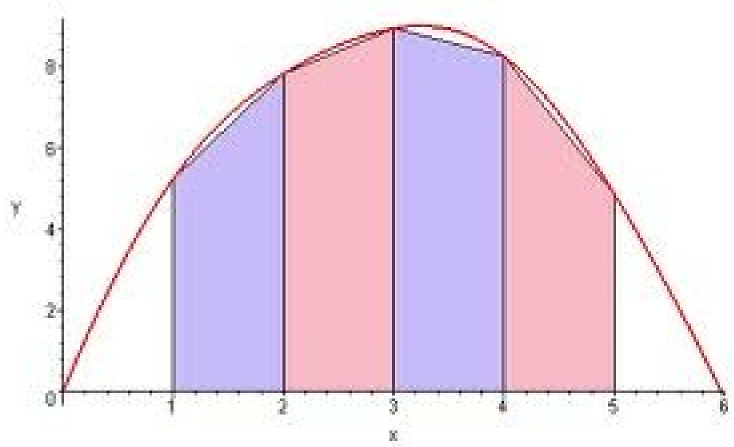




Work@class

•

$$\pi = \int_{0}^{1} \frac{4}{(1+x^2)} dx$$



Example: Compute PI (1)

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

int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, I, rc;
  double PI25DT = 3.141592653589793238462643;
  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);
  while (!done)
  {
      if (myid == 0)
      {
      printf(“Enter the number of intervals: (0 quits) “);
          scanf(“%d”, &n);
      }
      MPI_BCAST(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0)
  }
```

Example: Compute PI (2)

```
    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_COMM_WORLD);

    if (myid == 0) printf(“pi is approximately %.16f, Error is
%.16f\n”, pi, fabs(pi – PI25DT));

    MPI_Finalize();
    return 0;
}
```

Profiling Support: PMPI

- Profiling layer of MPI
- Implemented via additional API in MPI library
  - Different name: PMPI_Init()
  - Same functionality as MPI_Init()
- Allows user to:
  - define own MPI_Init()
  - Need to call PMPI_Init():
- User may choose subset of MPI routines to be profiled
- Useful for building performance analysis tools
  - Vampir: Timeline of MPI traffic (Etnus, Inc.)
  - Paradyn: Performance analysis (U. Wisconsin)
  - mpiP: J. Vetter (LLNL)
  - ScalaTrace: F. Mueller et al. (NCSU)

```
MPI_Init(…) {
  collect pre stats;
  PMPI_Init(…);
  collect post stats;
}
```

When to use MPI

- Portability and Performance
- Irregular data structure
- Building tools for others
- Need to manage memory on a per processor basis

Summary

- The parallel computing community has cooperated on the development of a standard for message-passing libraries.
- There are many implementations, on nearly all platforms.
- MPI subsets are easy to learn and use.
- Lots of MPI material is available.

Para Observar y Ejecutar

- http://people.sc.fsu.edu/~jburkardt/cpp_src/mpi/mpi.html

  https://computing.llnl.gov/tutorials/mpi/