Introducción a la programación en paralelo

Francisco Almeida y Francisco de Sande

Departamento de Estadística, I.O. y Computación

Universidad de La Laguna

La Laguna, 7-8 de septiembre de 2005

ULL Universidad de La Laguna

El Modelo de Paso de Mensajes

Francisco Almeida y Francisco de Sande

Departamento de Estadística, I.O. y Computación

Universidad de La Laguna

La Laguna, 7-8 de septiembre de 2005

Outline

- Introduction – Background
- Communications under the message passing model
  - Basic operations: send / receive
    - Synchronous/Asynchronous operations
    - Blocking/non-Blocking operations
    - Buffered/non-buffered operations
  - Collective Operations
- Programming Models
- The MPI Message Passing Library
- Applications

Introducción: el profesor

Información personal:

Francisco Almeida *aka* Paco

falmeida@ull.es

Despacho no. 82, 4ª Planta Física-Matemáticas

Tel. 922.318.173

Introducción: el profesor

Formación académica:

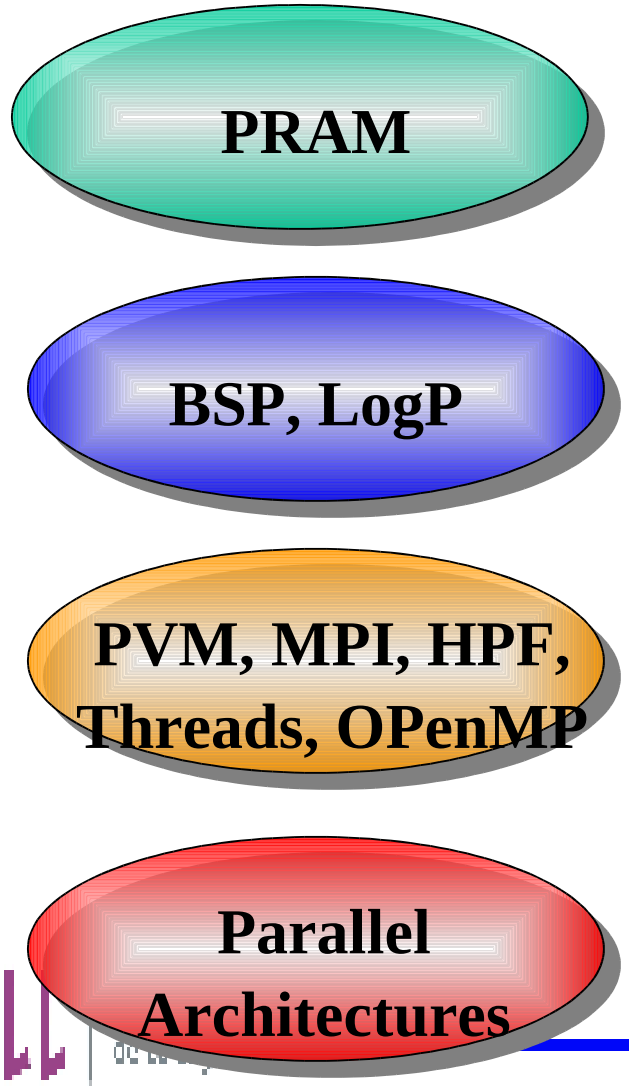
Lic. en Ciencias Matemáticas (1988)

Doctor en Informática (1996)

Campo de trabajo:

Computación de altas Prestaciones (MPI, OpenMP, lenguajes, compiladores, algoritmos)

The Parallel Model



Computational Models

Programming Models

Architectural Models

Shared Memory Architectures

All processors have access to one global memory

Key feature is a *single address space* across the whole memory system

- Every processor can read and write all memory locations, and processors communicate by reading/writing to the global memory
- One logical memory space

Processor caches are kept coherent

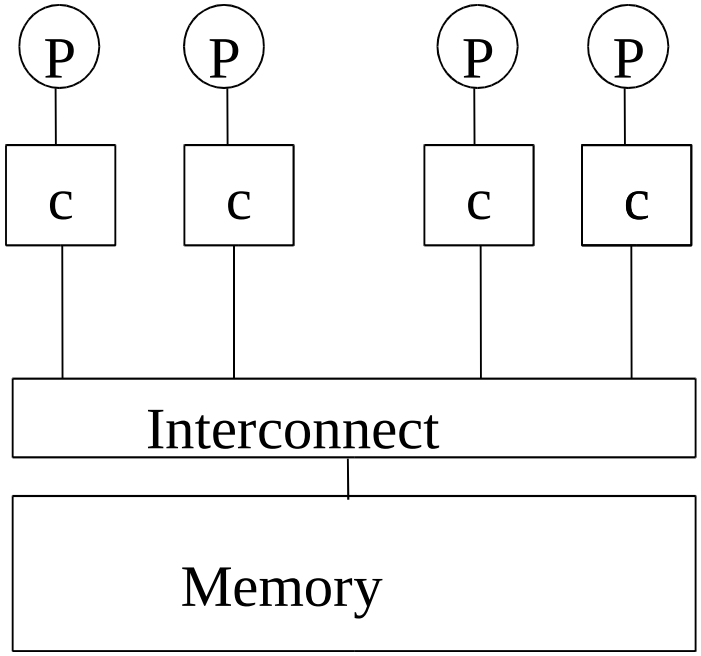
- All processors have same view of memory

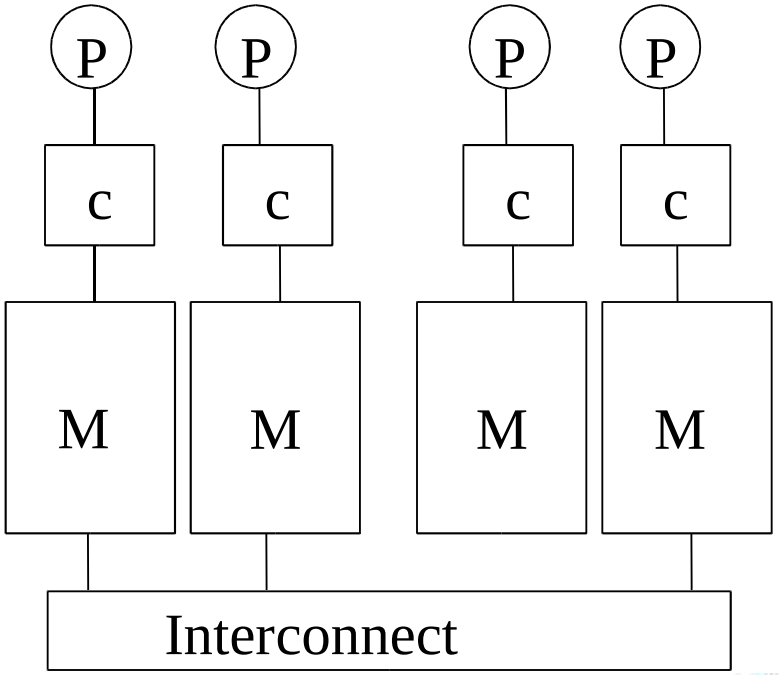
Examples: Sun HPC10000, multiprocessor PCs, Cray SV1, Compaq ES40, HP V class

Shared Memory Architectures

Two main types:

- True shared memory
- *Distributed Shared Memory (DSM)*





True Shared Memory

Distributed Shared Memory

Distributed memory architecture

Each processor has its own memory

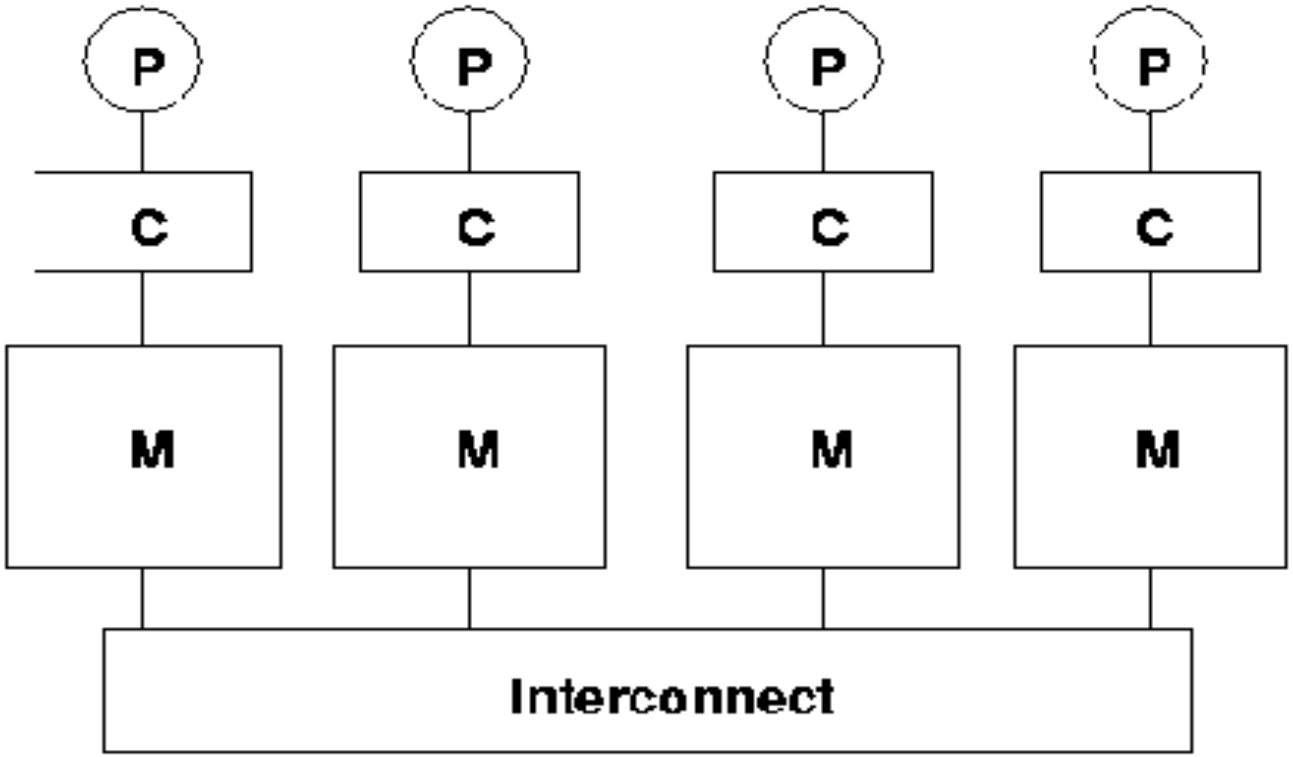
Each node (processor/memory pair) has separate address space.

Each node runs a separate copy of the OS

Nodes (processor/memory pairs) communicate through a network (dedicated, high bandwidth, low latency)

Examples: Beowulf clusters, Cray T3E

Distributed memory architecture



Distributed memory

SMPs

In true shared memory systems, also called *symmetric multiprocessors* or *SMPs*, the interconnect is often a bus.

It makes cache coherency simple, but doesn't scale to large numbers of processors.

The memory access time from any processor is approximately the same for any part of main memory. These are UMA (uniform memory access) machines.

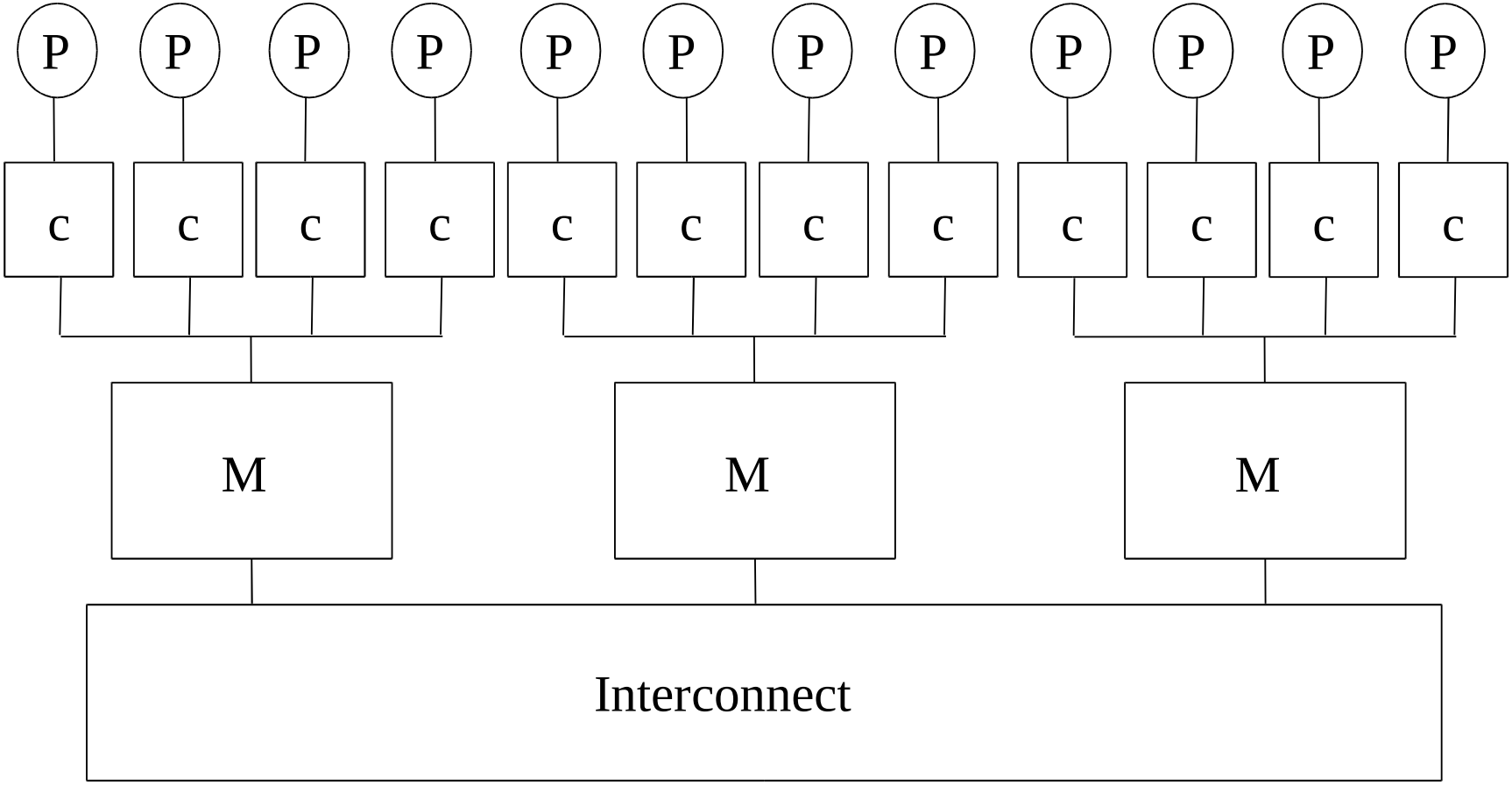
Clustered architectures

Many current systems consist of clusters of shared memory machines

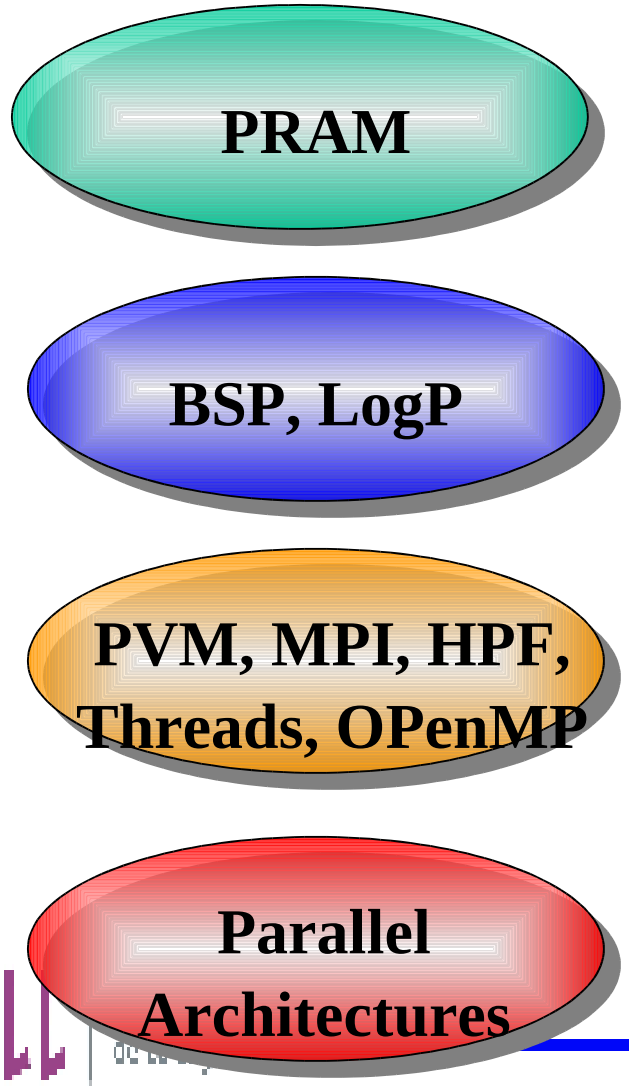
Some have separate address spaces on each shared memory node - *clustered SMPs* (e.g. IBM SP, NEC SX-5, Compaq SC)

Others have a single address space across the whole machine - *distributed shared memory* or *ccNUMA* (e.g. SGI Origin 3000)

Most DSM systems are clustered:



The Parallel Model



Computational Models

Programming Models

Architectural Models

Parallel Programming Models

Parallel programming models exist as an abstraction above hardware and memory architectures.

These models are NOT specific to a particular type of machine or memory architecture.

Any of these models can (theoretically) be implemented on any underlying hardware.

Which model to use is often a combination of what is available and personal choice.

There is no "best" model, although there certainly are better implementations of some models over others.

Drawbacks that arise when solving Problems using Parallelism

Parallel Programming is more complex than sequential.

Results may vary as a consequence of the intrinsic non determinism.

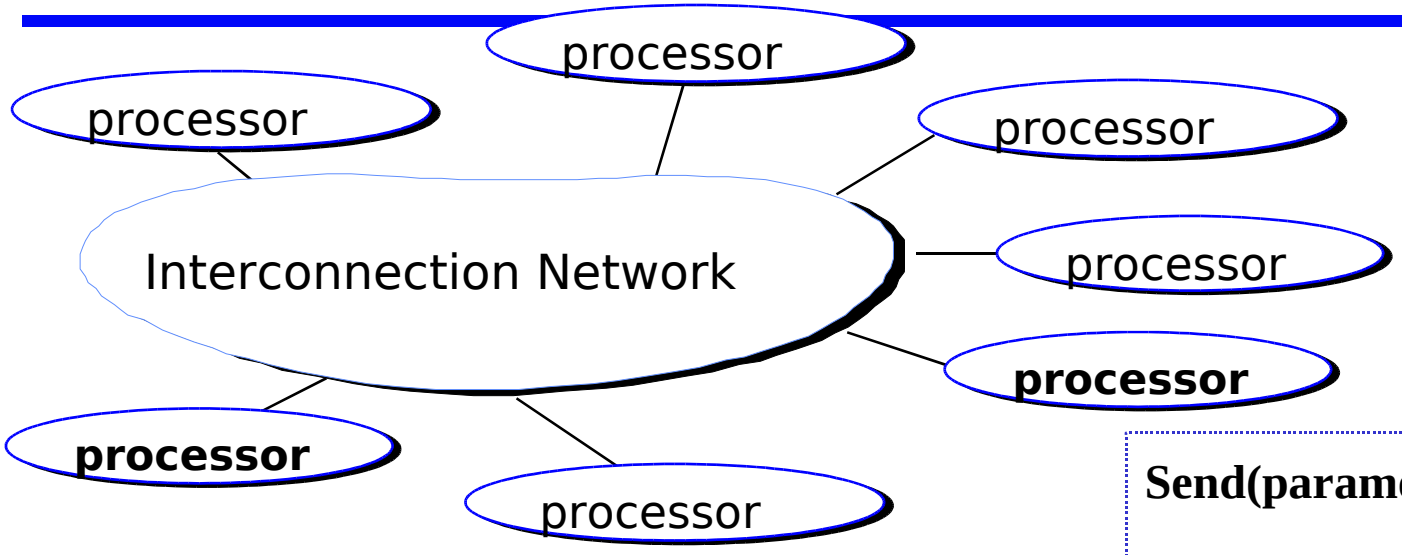
New problems. Deadlocks, starvation...

Is more difficult to debug parallel programs.

Parallel programs are less portable

The Message Passing Model

The Message Passing Model



Send(parameters)

Recv(parameters)

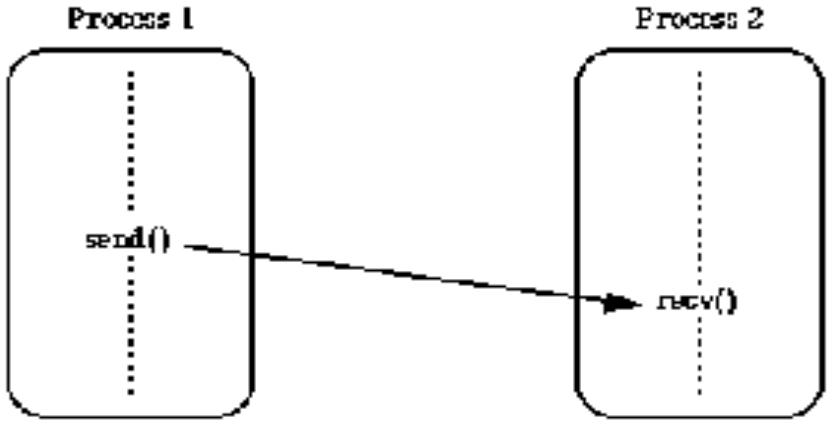


Figure 2.2 Passing a message between processes using send() and recv() system calls

Process vs Threads

Process - Task

A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.

Independent parallelizable subparts of a problem.

Thread

Lightweight process

Communications

Communication

Parallel processes typically need to exchange data.

There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.

Communications

The need for communications between processes depends upon your problem:

You DON'T need communications

Some types of problems can be decomposed and executed in parallel with virtually no need for processes to share data.

These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-processes communication is required.

You DO need communications

Most parallel applications are not quite so simple, and do require processes to share data with each other.

Efficiency of communications

Very often, the programmer will have a choice with regard to factors that can affect communications performance.

Which implementation for a given model should be used?

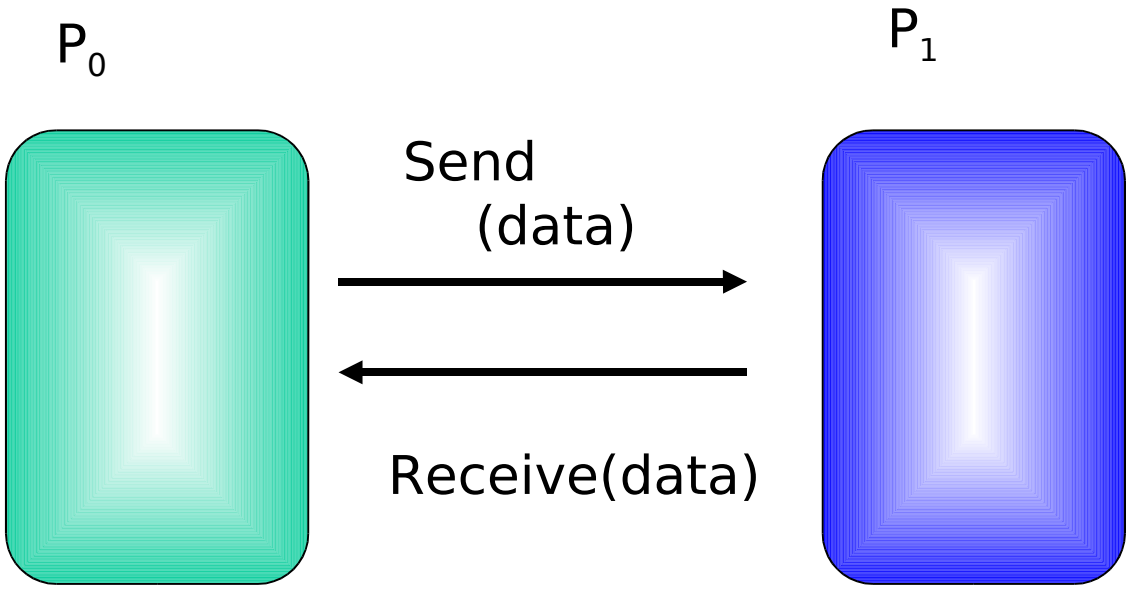
> Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.

What type of communication operations should be used?

> Asynchronous communication operations can improve overall program performance.

Network media - some platforms may offer more than one network for communications. Which one is best?

Communications: Ping-Pong



Maoui/Torke

```
qsub    -l    nodes=NODES_DESCRIPTION,walltime=TIME_LIMIT
  SUBMIT_FILE


-l nodes=16         ->  a process in each node
-l nodes=4:ppn=4    ->  4 nodes, 4 processors in each node
-l nodes=16:ppn=4   ->  this is the maximum,
                          4 processsors in each node.
walltime=01:00:00        -> duration of 1 hour
walltime=03:00:00:00     -> duration of 3 days
```

Maoui/Torke

```
#!/bin/sh
    # This finds out the number of nodes we have
    NP=$(wc -l $PBS_NODEFILE | awk '{print $1}')
        cd $PBS_O_WORKDIR
        # Make the MPI call
        mpirun -np $NP ./short_mpi
```

ex0/readme.txt

Communication Parameters

Latency vs. Bandwidth

latency is the time it takes to send a minimal (0 byte) message from point A to point B.

bandwidth is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.

Per-word ***transfer time***: inversely proportional to the available bandwith among the processors.

The linear model: $\beta + \tau n$

The cost of a communication can be modeled by the linear model: $\beta + \tau n$

β: Latency between processors

τ: Per-word transfer time

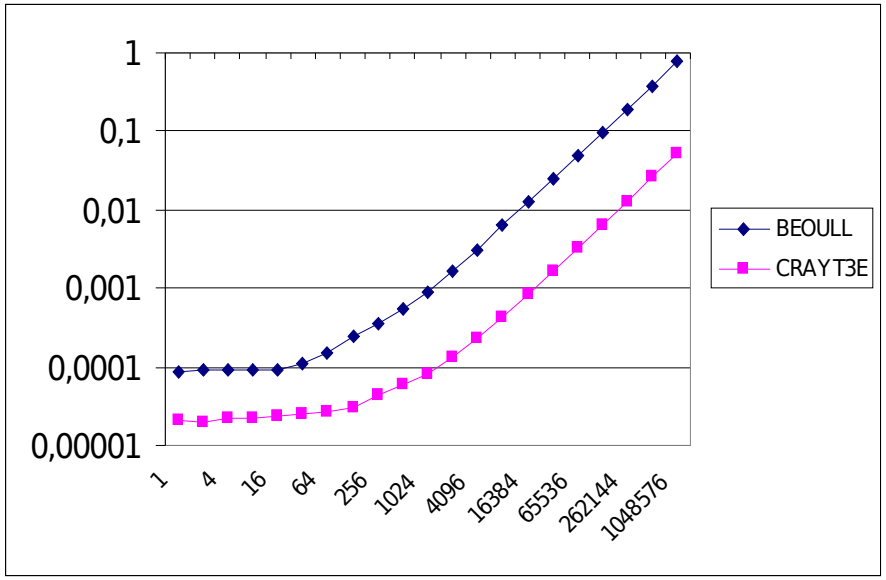
n: number of bytes to be sent

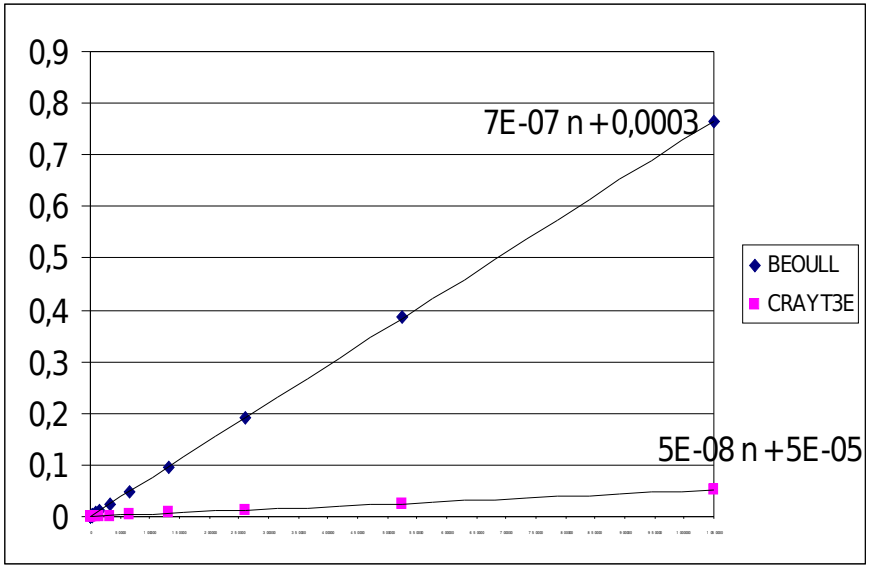
β and τ are measured using a ping-pong experiment (MPI_Send and MPI_Receive)

Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

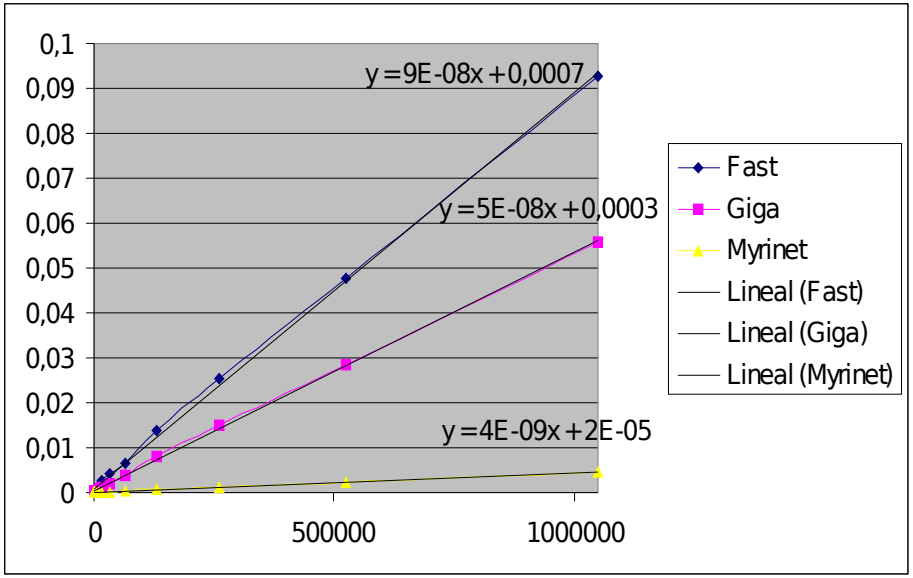
Linear Model to Predict Communication Performance

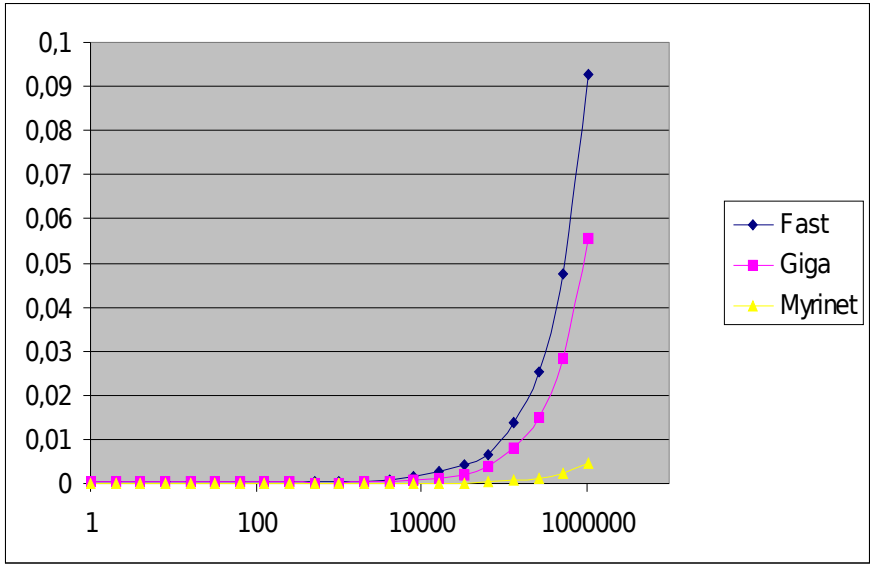
Time to send N bytes= τ n + β



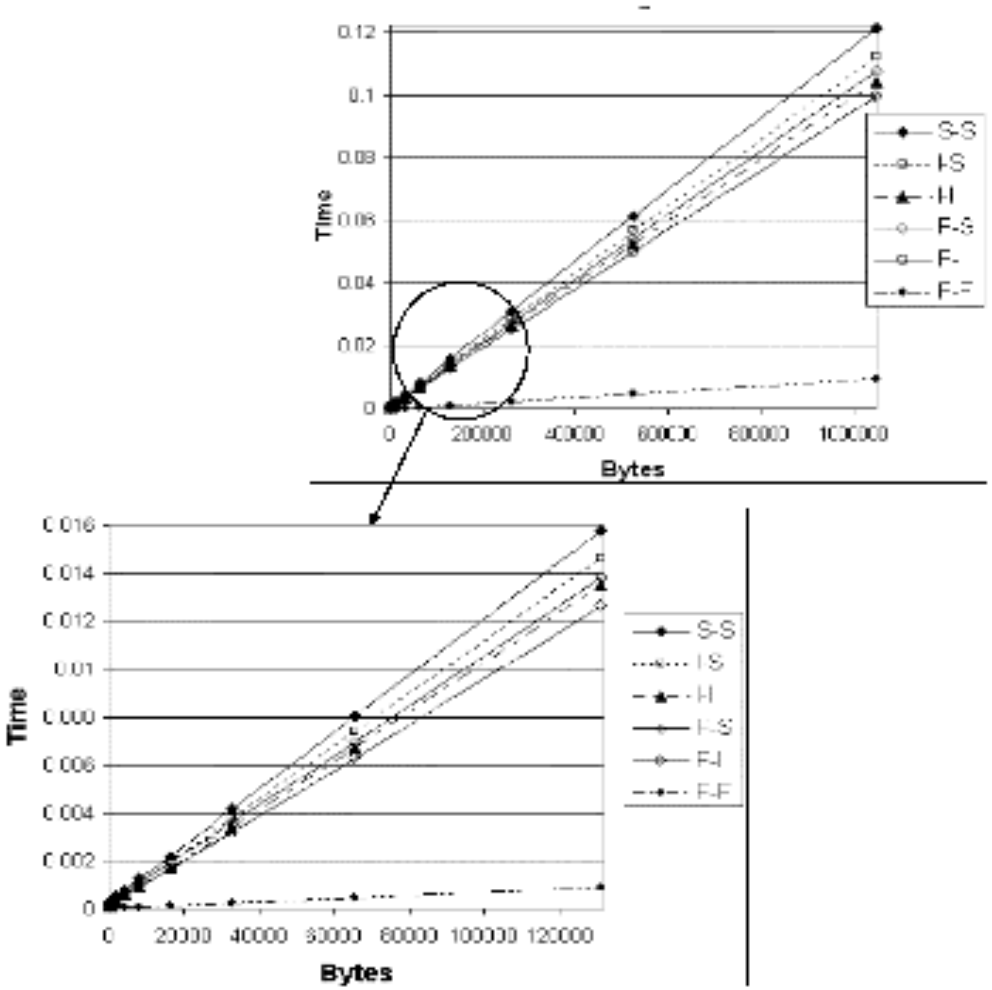


Communication Performace: Fast, Gigabit Ethernet, Myrinet

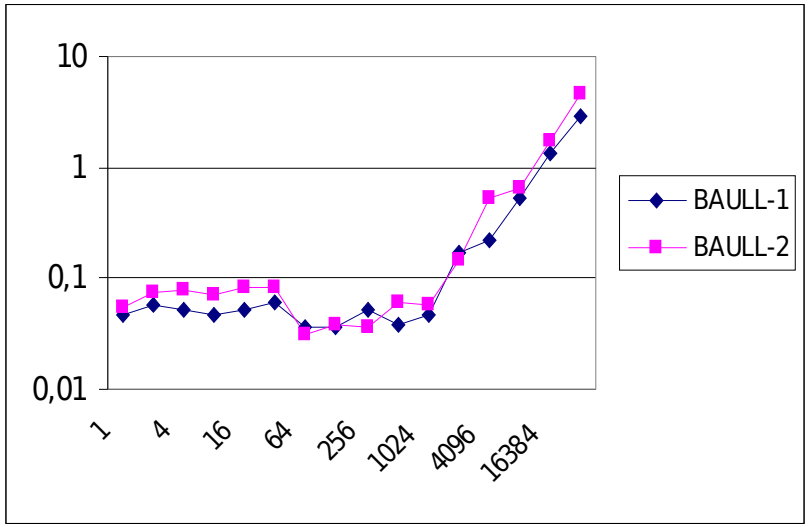


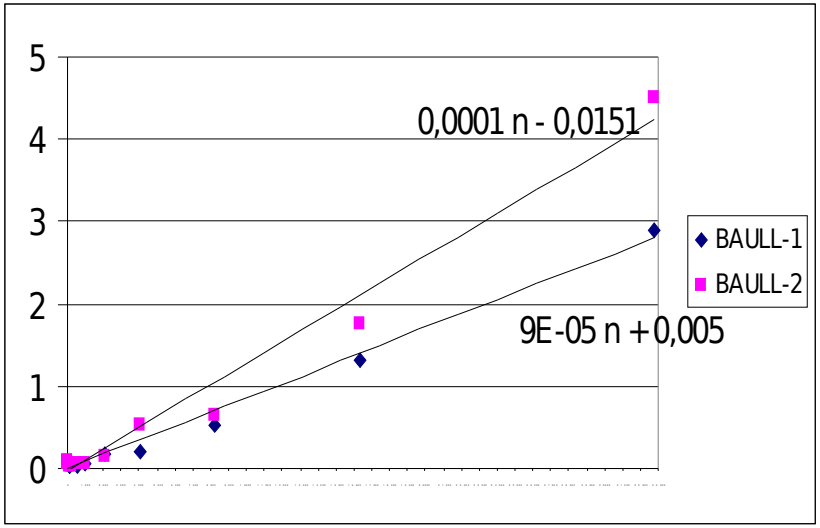


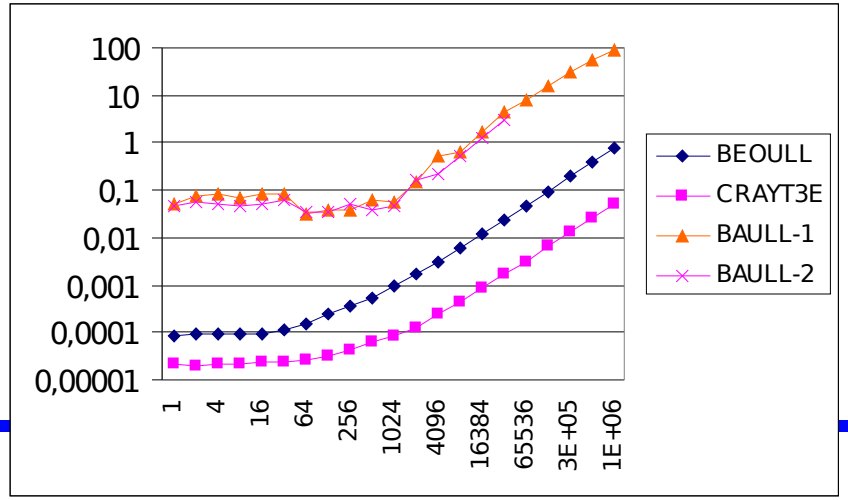
Communication Performance: Heterogeneous Cluster



Performace Prediction
BA - ULL







Communication Overhead

There are a number of important factors to consider when designing your program's inter-process communications:

Cost of communications

Inter-process communication virtually always implies overhead.

Machine cycles and resources that could be used for computation are instead used to package and transmit data.

Communications frequently require some type of synchronization between processes, which can result in tasks spending time "waiting" instead of doing work.

Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

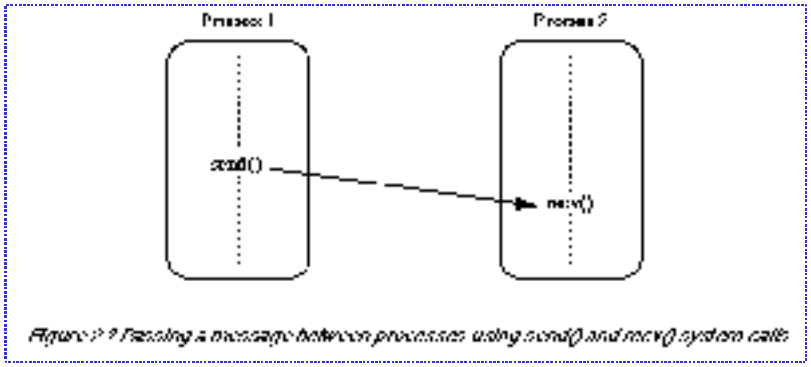
Communications

Synchronization

The coordination of parallel processes in real time, very often associated with communications.

Often implemented by establishing a synchronization point within an application where a process may not proceed further until another process(es) reaches the same or logically equivalent point.

Synchronization usually involves waiting by at least one processes, and can therefore cause a parallel application's wall clock execution time to increase.



Data dependence

A[j+1] = A[j] + z;

Communications

When designing parallel programs, communications must be carfully managed

SPMD Model

Single Program Multiple Data (SPMD):

SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

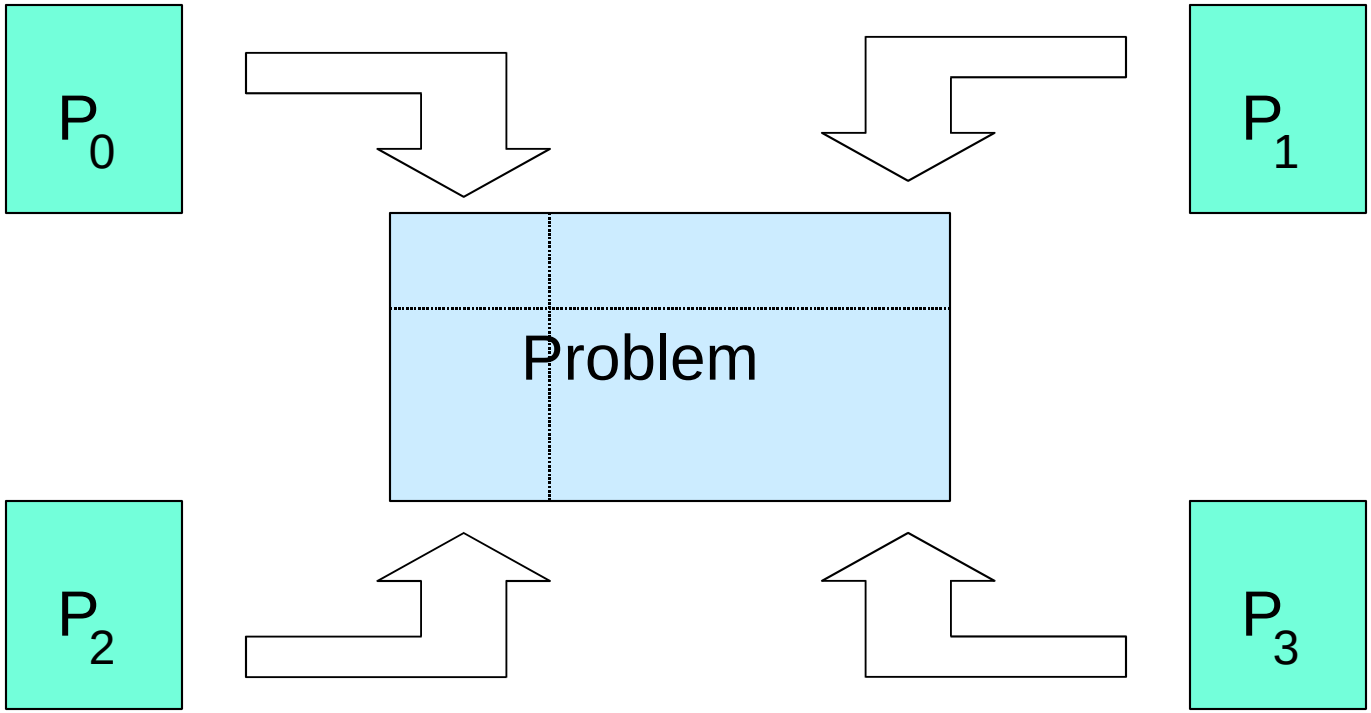
A single program is executed by all process simultaneously.

At any moment in time, tasks can be executing the same or different instructions within the same program.

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, processes do not necessarily have to execute the entire program - perhaps only a portion of it.

All processes may use different data

SPMD Model



Matrix Product

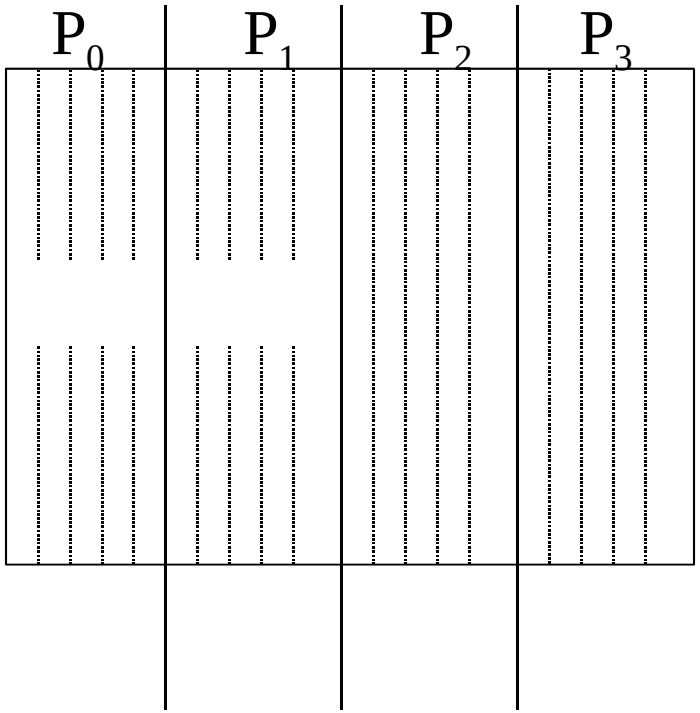
C = A * B

SPMD

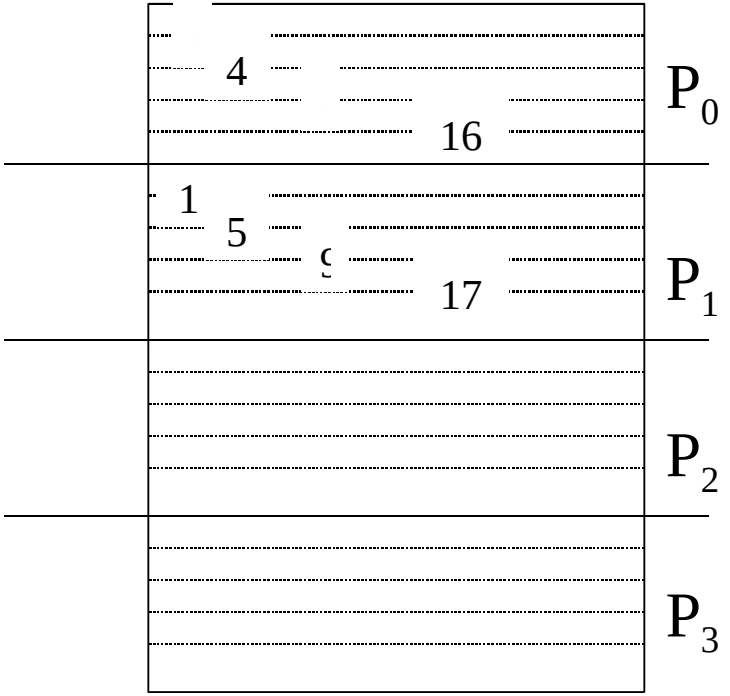
- Broadcast B
- Rowwise stripping A
- Local Products
- Collect Results

Data Distribution

Columnwise block striping



Rowwise cyclic striping



MPMD Model

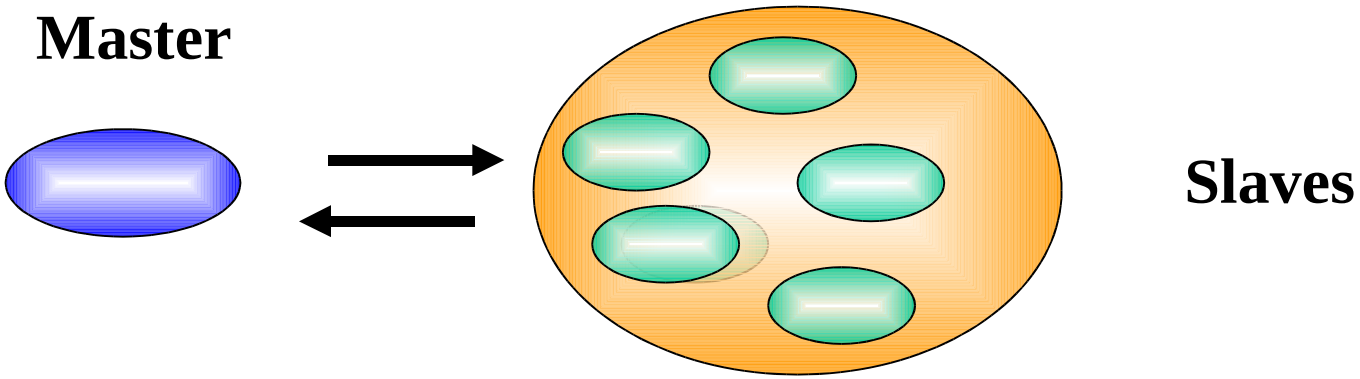
Multiple Program Multiple Data (MPMD):

Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

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.

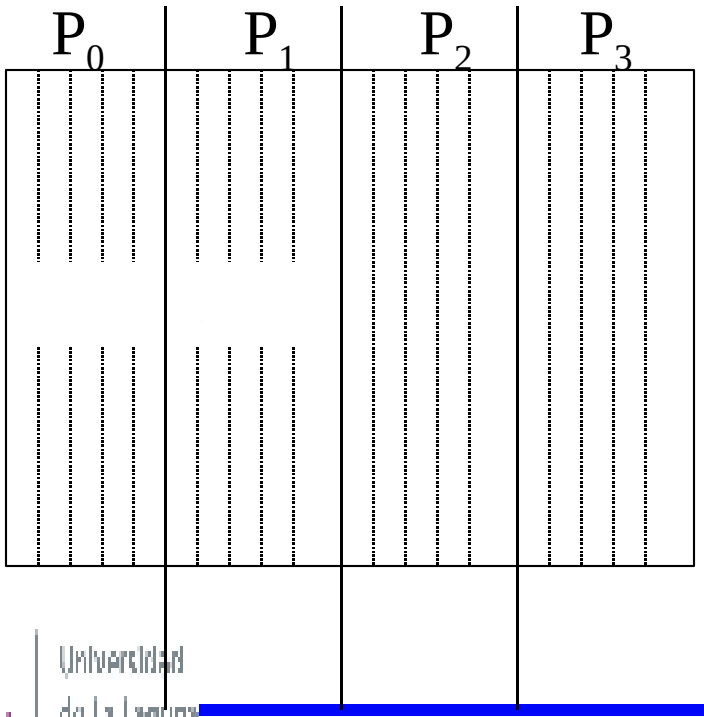
All tasks may use different data

The Master Slave Paradigm

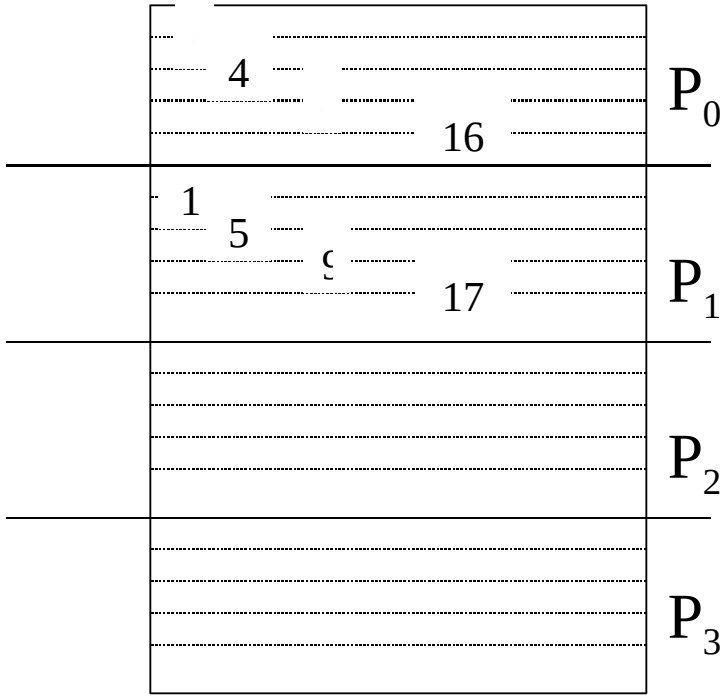


Matrix Product

Columnwise block striping



Rowwise cyclic striping



The Master-Slave Paradigm

The work W can be divided into a set *p* of independent task of arbitrary size $work_1$ ... $work_p$, that can be processed in parallel by the slave processors *1...p*.

The master divides *W* and distributes $work_i$ to processor *i*.

Slave *i* computes $work_i$ and returns the results to the master.

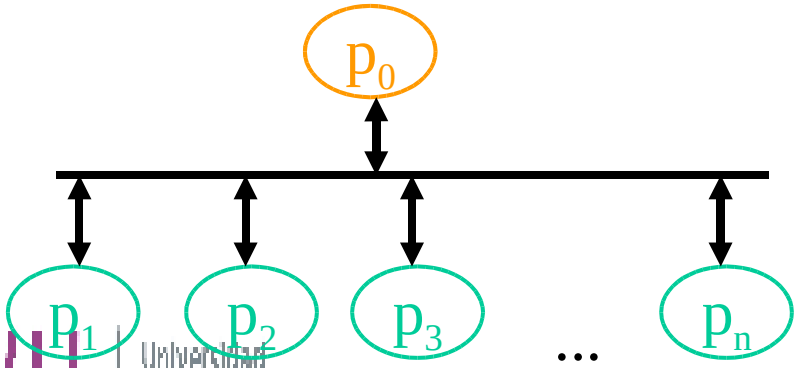
The master and the slaves are connected through a network. At a given time-step, at most one slave can communicate with the master.

Master Processor

```
for (proc = 1; proc <= p; proc++)
    send(proc, work[proc]);
for (proc = 1; proc <= p; proc++)
    receive(proc, result[proc]);
```

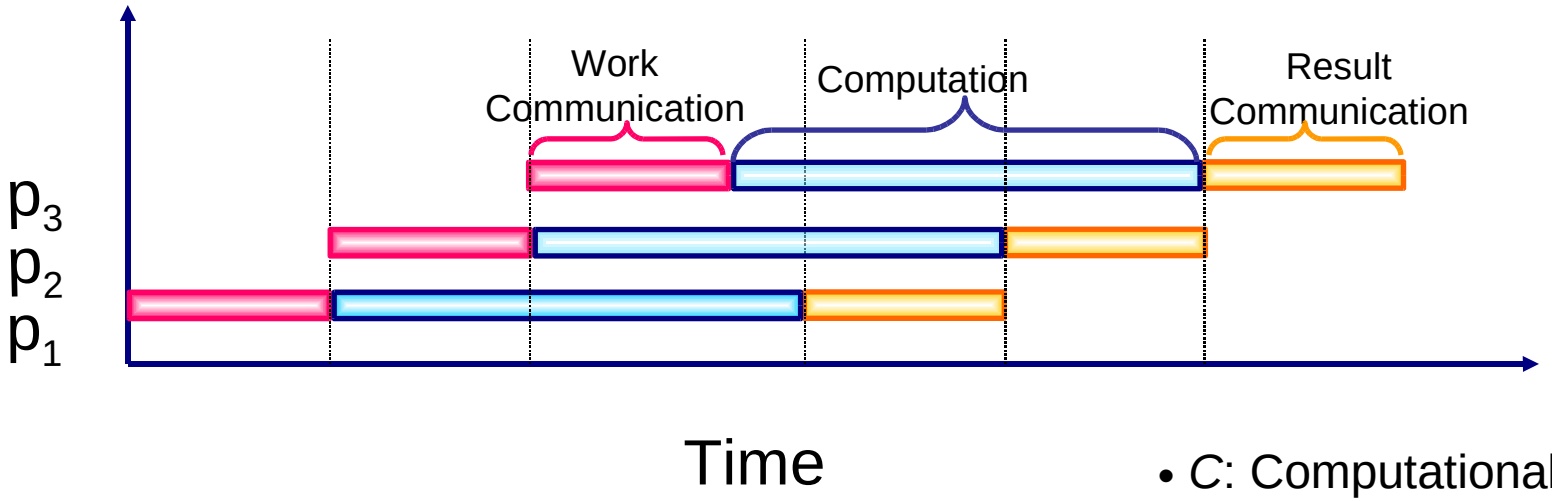
Slave Processor

```
receive(master, work);
compute(work, result);
send(master, result);
```



Master-Slave

- All the processors are identical. The optimal distribution of work is obtained when all the processor receive the same amount of work



- *C*: Computational Time
- *w*: size of a task
- *s*: size of a result

Running Time = $p(\beta + \tau w) + C + \beta + \tau s$

Send/Receive Operations

Send and Receive Operations

```
Send(void *sendbuf, int nelems, int dest);
Receive(void *recvbuf, int nelems, int source);
```

P_0

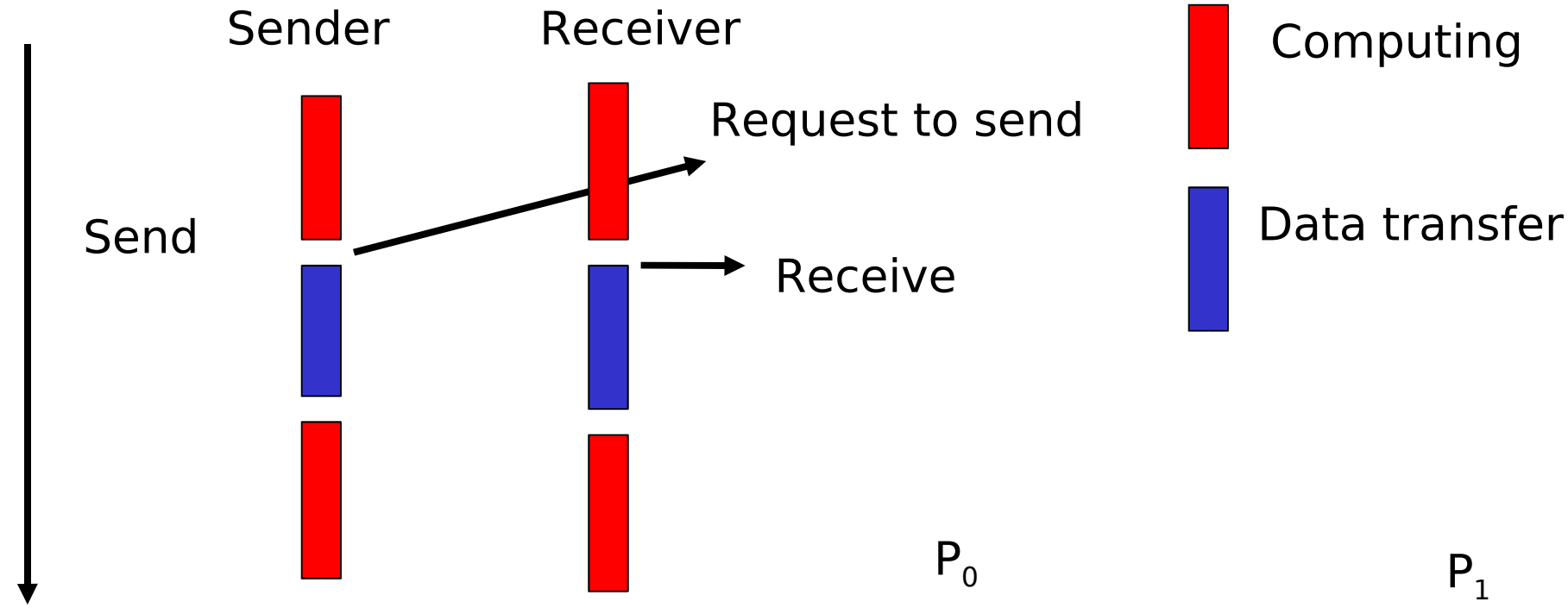
```
A = 100;
Send(&A, 1, 1);
A = 0;
```

P_1 P_1 depends on Value A

```
Receive(&A, 1, 0);
printf("%d\n", A);
```

Blocking Operation: The value received by process P_1 must be 100

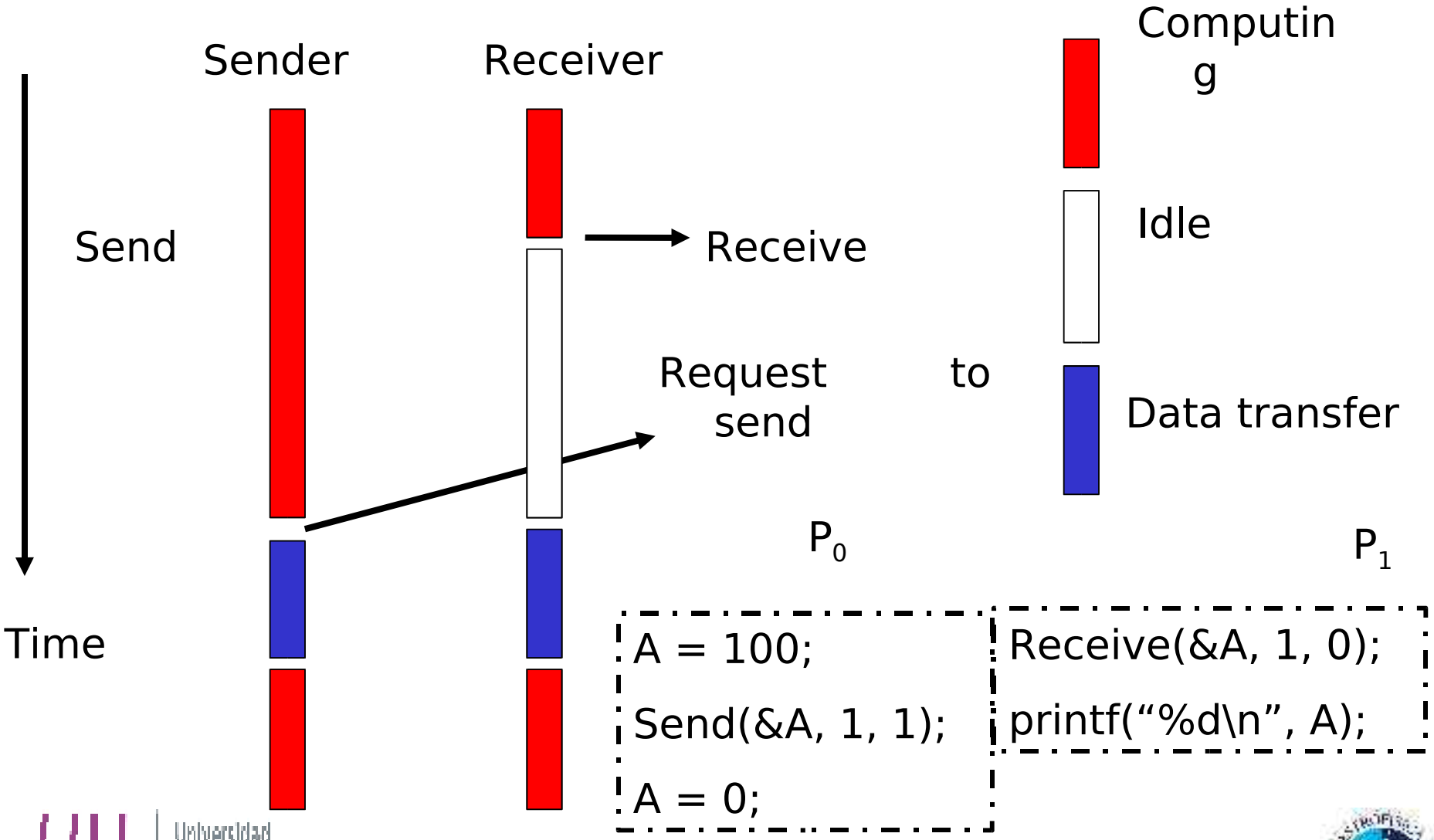
Sender and Receiver come at the Same time



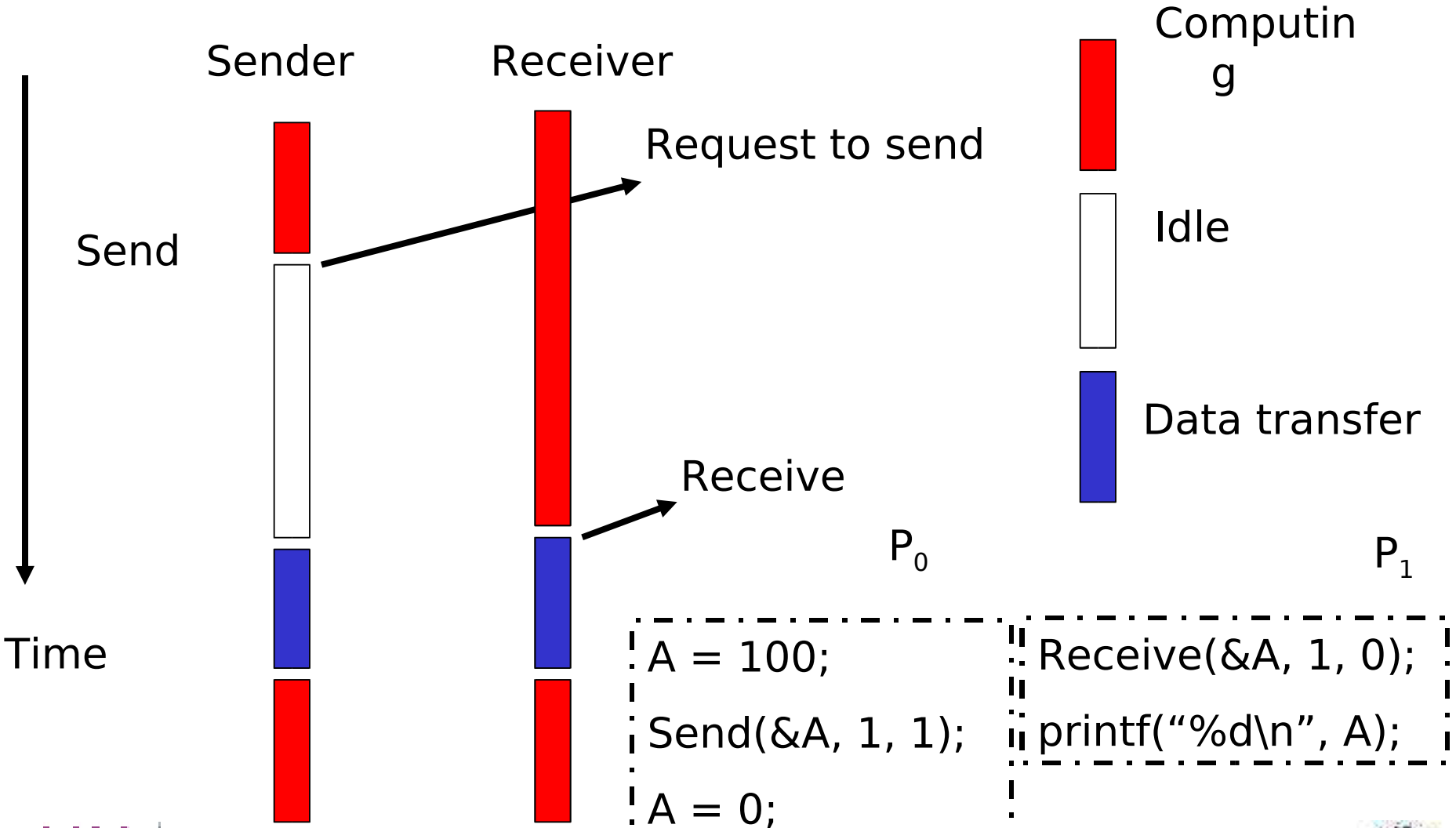

```
A = 100;
Send(&A, 1, 1);
A = 0;
```

```
Receive(&A, 1, 0);
printf("%d\n", A);
```

The Receiver Comes First



The Sender Comes First



Blocking Non-Buffered

```
Send(void *sendbuf, int nelems, int dest);

Receive(void *recvbuf, int nelems, int source);
```

P_0

```
A = 100;

Send(&A, 1, 1);

Receive(&B, 1, 1);
```

P_1

```
B = 200;

Send(&B, 1, 0);

Receive(&A, 1, 0);
```

Blocking Non-Buffered

```
Send(void *sendbuf, int nelems, int dest);

Receive(void *recvbuf, int nelems, int source);
```

P_0

```
A = 100;

Send(&A, 1, 1);

Receive(&B, 1, 1);
```

P_1

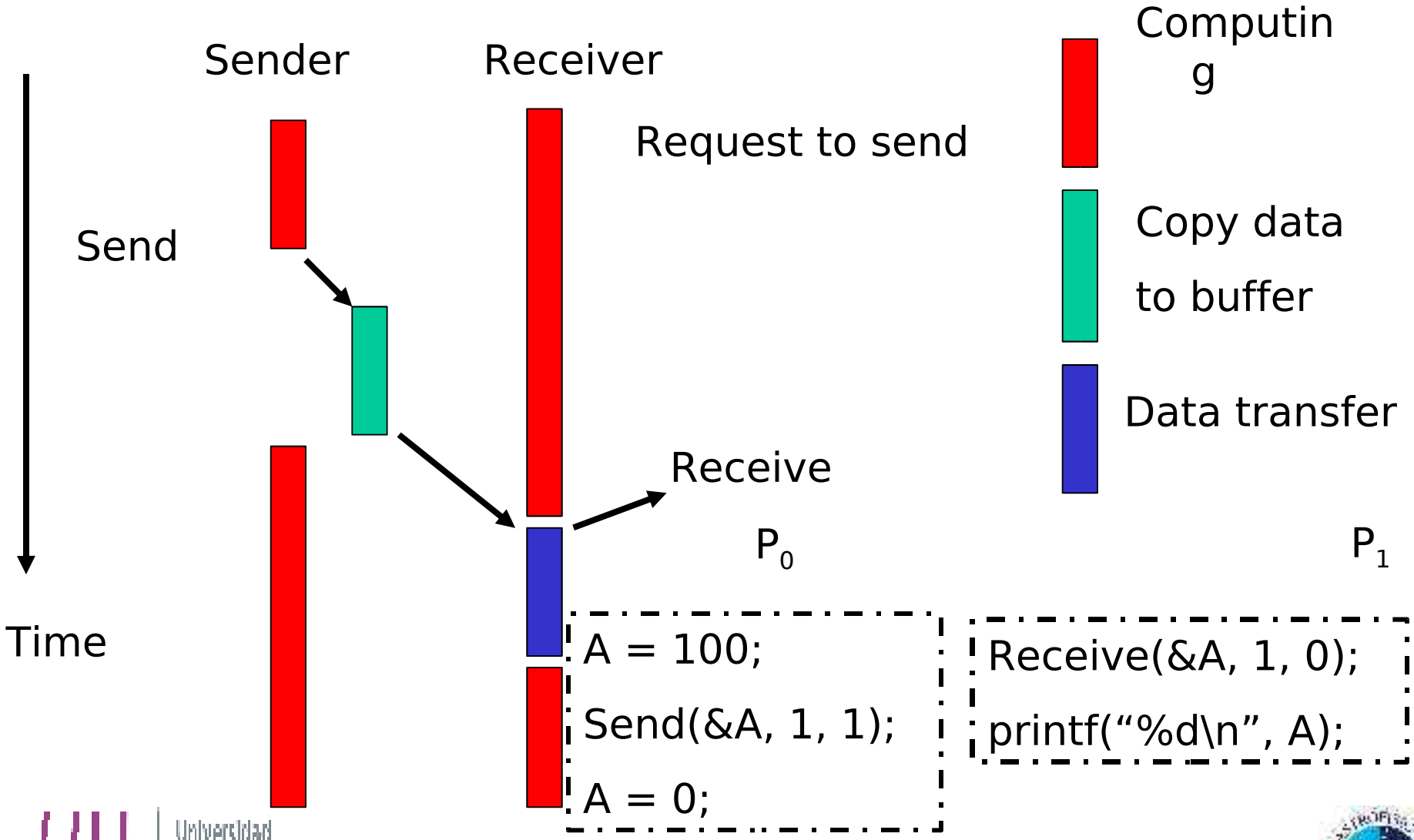
```
B = 200;

Send(&A, 1, 0);

Receive(&A, 1, 0);
```

DEADLOCK

The Sender Comes First: Blocking Buffered



Synchronous vs. asynchronous communications

Synchronous communications require some type of "handshaking" between processes that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.

Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.

Asynchronous communications allow processes to transfer data independently from one another. For example, processes 1 can prepare and send a message to task 2, and then immediately begin doing other work. When processes 2 actually receives the data doesn't matter.

Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.

Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Types of Synchronization

Barrier

Usually implies that all tasks are involved

Each task performs its work until it reaches the barrier. It then stops, or "blocks".

When the last task reaches the barrier, all tasks are synchronized.

What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Synchronous communication operations

Involves only those tasks executing a communication operation

When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

The Message Passing Model

Implementations

Implementations

From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code.

The programmer is responsible for determining all parallelism.

Historically, a variety of message passing libraries have been available since the 1980s.

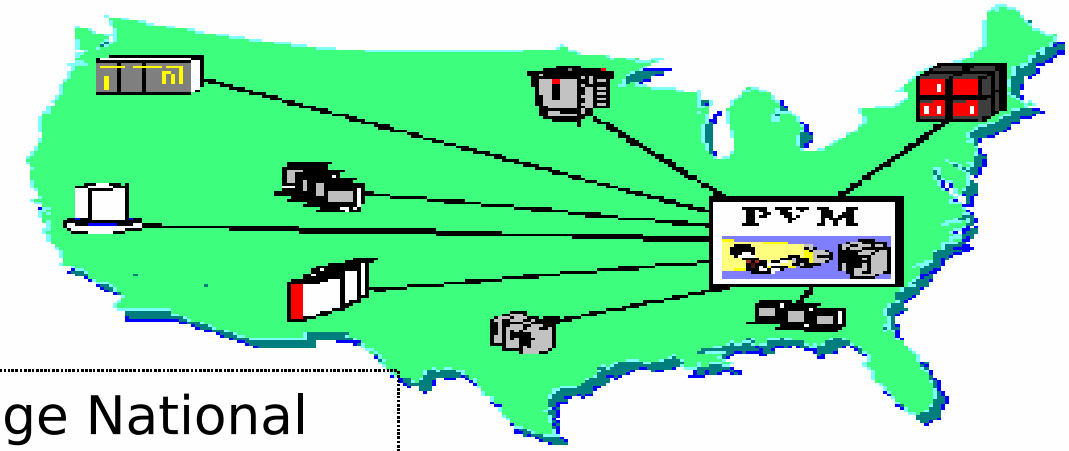
These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

Standard libraries: PVM, MPI.

Vendors' proprietary libraries.

PVM

Heterogeneous Distributed Computing

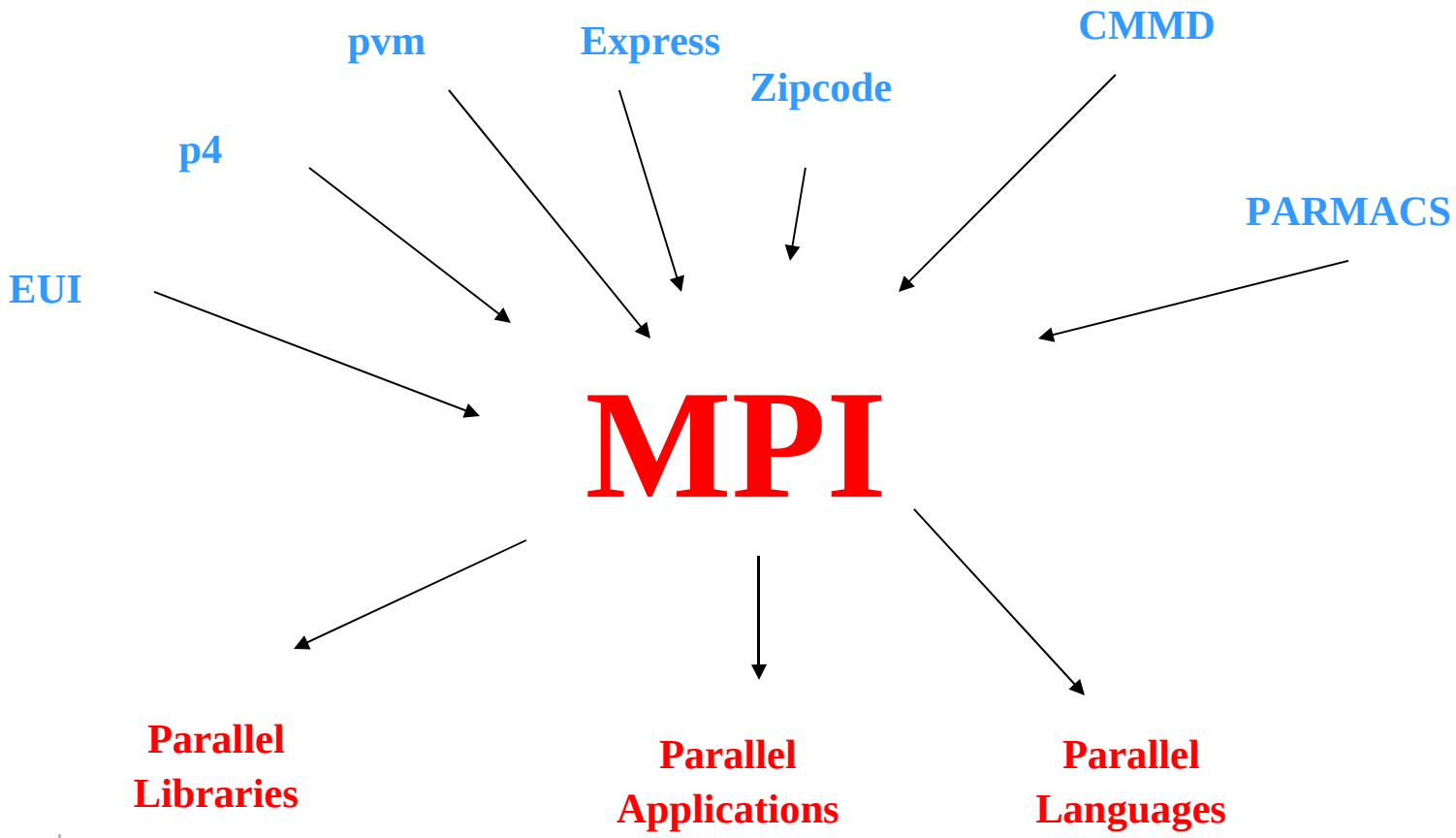


1989 Oak Ridge National

- PVM is a software package that permits a heterogeneous collection of serial, parallel, and vector computers hooked together by a network to appear as one large computer.
- PVM supplies C and Fortran routines for asynchronous message-passing and process control.
- PVM is popular because it is small, portable, and easy to install and use.
- Being used around the world to develop small and large scale scientific applications.

MPI

http://www-unix.mcs.anl.gov/mpi/



MPI

- What Is MPI?
- Message Passing Interface standard
- The first standard and portable message passing library with good performance
- "Standard" by consensus of MPI Forum (1992) participants from over 40 organizations
- Finished and published in May 1994, updated in June 1995. In 1997 MPI-2 was released.
- Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html

MPI

- What does MPI offer?

- Standardization - on many levels replacing virtually all other message passing implementations used for production work.

- Portability - to existing and new systems. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. Very few have a full implementation of MPI-2.

- Performance - comparable to vendors' proprietary libraries. For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.

- Richness - extensive functionality, many quality implementations

PVM - MPI

PMV

- DeFacto Standard
- Virtual Machine
- Heterogenous Computing, Interoperability
- Fault Tolerance
- Non Efficient Communications

MPI

- Standard by concensus
- Very Efficient communications
- Many point-to-point communications

MPICH - A Portable Implementation of MPI

MPICH is freely available and is distributed as source.

The Unix (all flavors) version of MPICH .

The Microsoft Windows version of MPICH.

MPICH-G2, the Globus version of MPICH.

MPICH supports a wide range of systems. These include workstation clusters, shared memory systems, and massively parallel supercomputers. A list of supported systems is available.

The MPICH implementation is not thread-safe.

William D. Gropp. Argonne National Laboratory

MPI Versions

LA-MPI: The Los Alamos Message Passing Interface

LA-MPI is an implementation of the Message Passing Interface (MPI) motivated by a growing need for **fault tolerance** at the software level in large high-performance computing (HPC) systems.

LA-MPI is no longer in active development, but is being maintained for use on production systems at LANL, and we wellcome other users.

LAM/MPI: Enabling Efficient and Productive MPI Development

LAM/MPI was designed to operate on **heterogeneous clusters**. With support for **Globus and Interoperable MPI**, **LAM/MPI can spawn clusters of clusters**.

The LAM team is comprised of students and faculty at **Indiana University**

Fault Tolerant MPI (FT-MPI).

Is a full 1.2 MPI specification implementation that provides process level **fault tolerance** at the MPI API level.

FT-MPI is built upon the fault tolerant HARNESS runtime system.

MPICH2

MPICH2 is an all-new implementation of MPI, designed to support research into high-performance implementations of MPI-1 and MPI-2 functionality.

In addition to the features in MPICH, MPICH2 includes support for one-side communication, dynamic processes, intercommunicator collective operations, and expanded MPI-IO functionality.

Clusters consisting of both single-processor and SMP nodes are supported. With the exception of users requiring the communication of heterogeneous data.

We strongly encourage everyone to consider switching to MPICH2.

Researchers interested in using using MPICH as a base for their research into MPI implementations should definitely use MPICH2.

OpenMPI: A High Performance Message Passing Library

Open MPI is a project combining technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI) in order to build the best MPI library available.

Will be released in the first quarter of 2005, and offers advantages for system and software vendors, application developers and computer science researchers.

Features implemented or in short-term development for Open MPI include:

- Full MPI-2 standards conformance
- Thread safety and concurrency
- Dynamic process spawning
- High performance on all platforms
- Reliable and fast job management
- Network and process fault tolerance
- Support data and network heterogeneity

OpenMPI: A High Performance Message Passing Library

- Single library supports all networks
- Run-time instrumentation
- Many job schedulers supported
- Many OS's supported (32 and 64 bit)
- Production quality software
- Portable and maintainable
- Tunable by installers and end-users
- Extensive user and installer guides
- Internationalized error messages
- Component-based design, documented APIs
- CPAN-like tool for component management
- Active, responsive mailing list
- Open source license based on the BSD license

http://www.open-mpi.org/

Format of MPI routines

C bindings

For C, the general format is

```
rc = MPI_Xxxxx(parameter, ... )
```

Case sensitive.

rc is a return code, and is type integer. Upon success, it is set to MPI_SUCCESS.

C programs should include the file "mpi.h". This contains definitions for MPI constants and functions.

Fortran bindings

For Fortran, the general format is

```
Call MPI_XXXXX(parameter,..., ierror)
```

Case insensitive

```
call mpi_xxxxx(parameter,..., ierror)
```

Instead of the function returning with an error code, as in C, the Fortran versions of MPI routines usually have one additional parameter in the calling list, ierror, which is the return code. Upon success, ierror is set to MPI_SUCCESS.

Fortran programs should include 'mpif.h'. This contains definitions for MPI constants and functions.

MPI Routines

The basic outline of an MPI program follows these general steps:

Initialize for communications

Communicate to share data between processes

Exit in a "clean" fashion from the message-passing system when done communicating

MPI has over 125 functions. However, a beginning programmer usually can make do with only six of these functions. These six functions, illustrated and discussed in our sample program are:

Initialize for communications

MPI_INIT initializes for the MPI environment

MPI_COMM_SIZE returns the number of processes

MPI_COMM_RANK returns this process's number (rank)

Communicate to share data between processes

MPI_SEND sends a message

MPI_RECV receives a message

Exit from the message-passing system

MPI_FINALIZE

A Simple MPI Program

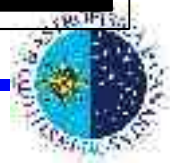
MPI hello.c

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char*argv[]) {
  int name, p, source, dest, tag = 0;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&name);
  MPI_Comm_size(MPI_COMM_WORLD,&p);
  printf("Hello from processor %d of %d\n",name, p);
  MPI_Finalize();
}
```

```
$> mpicc -o hello hello.c
$> mpirun -np 4 hello
Hello from processor 2 of 4
Hello from processor 3 of 4
Hello from processor 1 of 4
Hello from processor 0 of 4
```

```
$> mpif77 -o hello hello.f
$> mpirun -np 4 hello
Hello from processor 2 of 4
Hello from processor 3 of 4
Hello from processor 1 of 4
Hello from processor 0 of 4
```

ex1/readme



MPI Messages

MPI messages consist of two basic parts: the actual data that you want to send/receive, and an envelope of information that helps to route the data.

There are usually three calling parameters in MPI message-passing calls that describe the data, and another three parameters that specify the routing:

Message = data(3 parameters) + envelope(3 parameters)

startbuf,count,datatype, dest,tag,comm

\ | / \ | /

\--- DATA --/ ENVELOPE

MPI C Datatypes

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

MPI Fortran 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	

Envelope

The envelope provides information on how to match sends to receives.

Destination or source

These arguments are set to a rank in a communicator (see below). Ranks range from 0 to (size-1) where size is the number of processes in the communicator.

Destination is specified by the send and is used to route the message to the appropriate process.

Source is specified by the receive. Only messages coming from that source can be accepted by the receive call.

The receive can set source to **MPI_ANY_SOURCE** to indicate that any source is acceptable.

Envelope

Tag

An arbitrary number to help distinguish among messages.

The tags specified by the sender and receiver must match.

The receiver can specify **MPI_ANY_TAG** to indicate that any tag is acceptable.

Tag allows the program to distinguish between types of messages. Source simplifies programming. Instead of having a unique tag for each message, each process sending the same information can use the same tag.

Envelope

Communicators

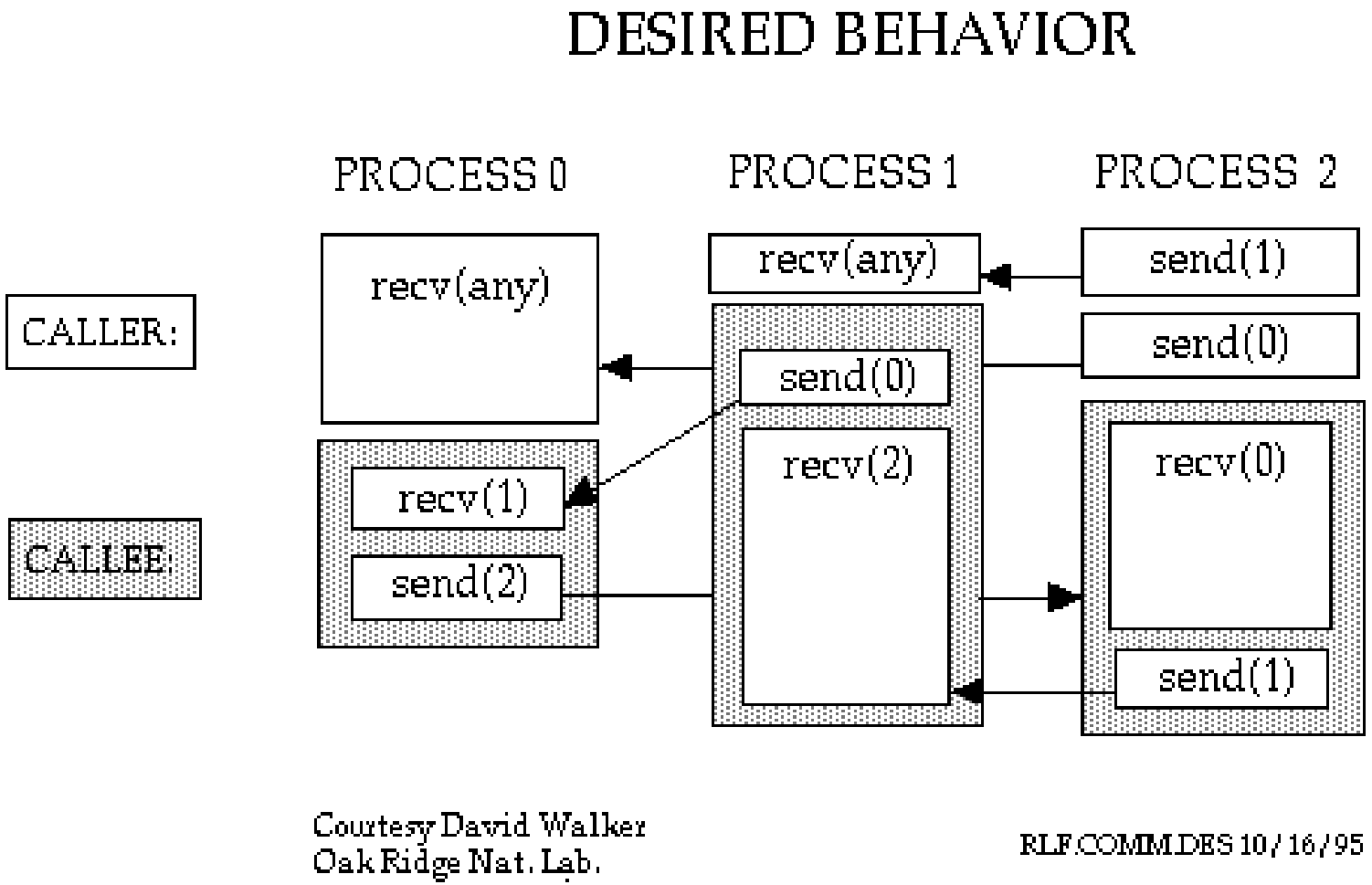
The communicator specified by the send must equal that specified by the receive.

A communicator defines a communication "universe", and that processes may belong to more than one communicator.

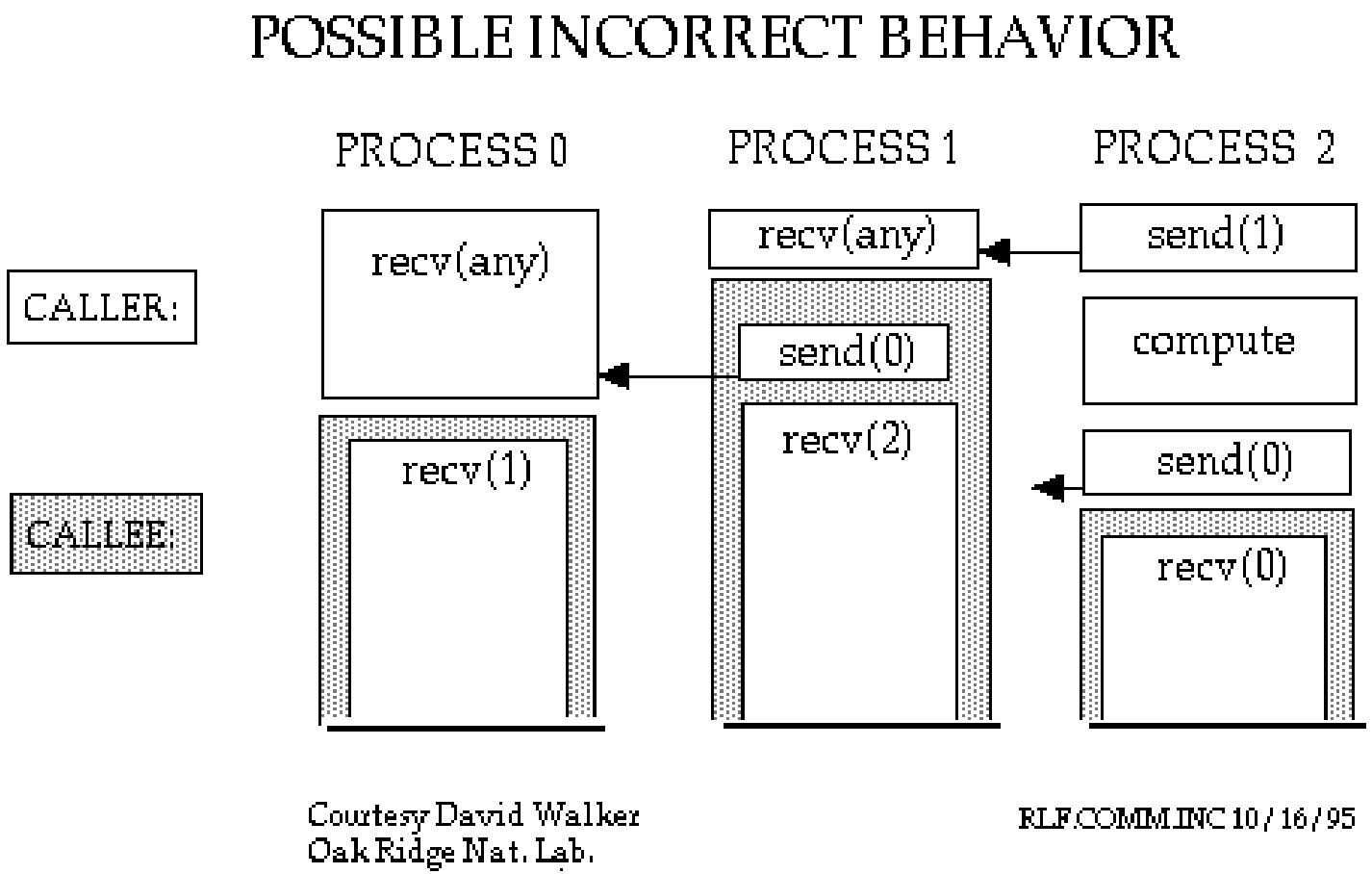
A predefined communicator **MPI_COMM_WORLD** includes all processes in the application.

Why have communicators?

Communicators



Communicators



A Simple MPI Program

MPI helloms.c

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"
main(int argc, char*argv[]) {

int name, p, source, dest, tag = 0;
char message[100];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&name);
MPI_Comm_size(MPI_COMM_WORLD,&p);
if (name != 0) {
  printf("Processor %d of %d\n",name, p);
  sprintf(message,"greetings from process %d!",
          name);
  dest = 0;
  MPI_Send(message, strlen(message)+1,
           MPI_CHAR, dest, tag,
           MPI_COMM_WORLD);
} else {
  printf("processor 0, p = %d ",p);
  for(source=1; source < p; source++) {
    MPI_Recv(message,100, MPI_CHAR, source,
             tag, MPI_COMM_WORLD, &status);
    printf("%s\n",message);
  }
}
MPI_Finalize();
}
```

```
$> mpirun -np 4 helloms
Processor 2 of 4
Processor 3 of 4
Processor 1 of 4
processor 0, p = 4
greetings from process 1!
greetings from process 2!
greetings from process 3!
```

MPI_Status

In C, status is a structure that contains three fields named:

MPI_SOURCE,

MPI_TAG

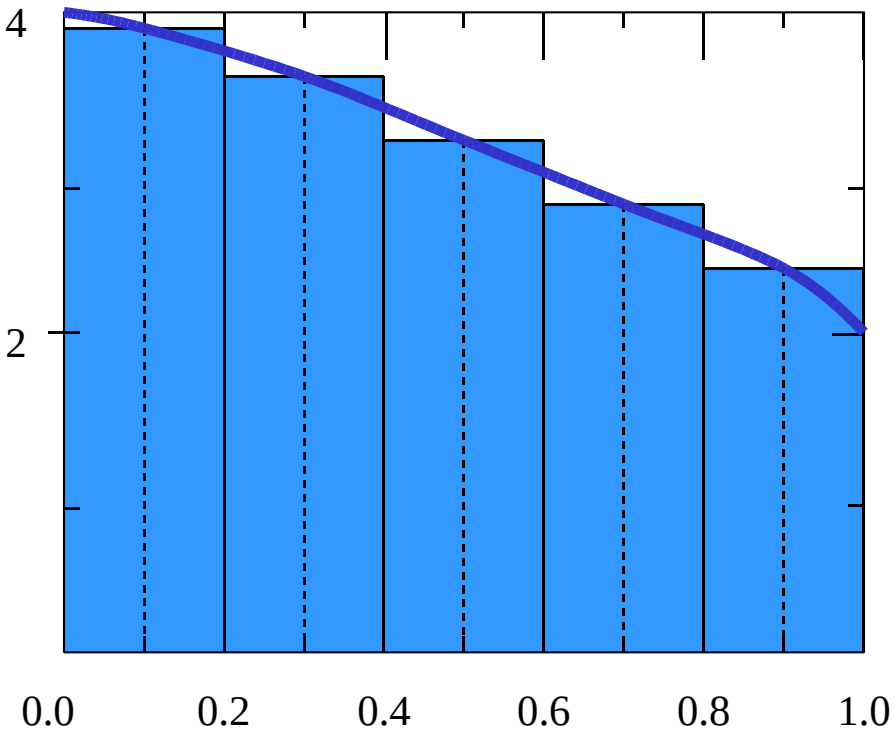
MPI_ERROR

the structure may contain additional fields.

Thus, status.MPI_SOURCE, status.MPI_TAG and status.MPI_ERROR contain the source, tag, and error code, respectively, of the received message.

Computing π: Sequential

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx$$




```
double t, pi=0.0, w;
long i, n = ...;
double local, pi = 0.0;
....

 h   = 1.0 / (double) n;
 for (i  = 0; i < n; i++) {
    x = (i + 0.5) * h;
    pi += f(x);
 }
 pi *= h;
```

ex4/readme.txt

Scope of communications

Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.

Point-to-point - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.

Collective - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

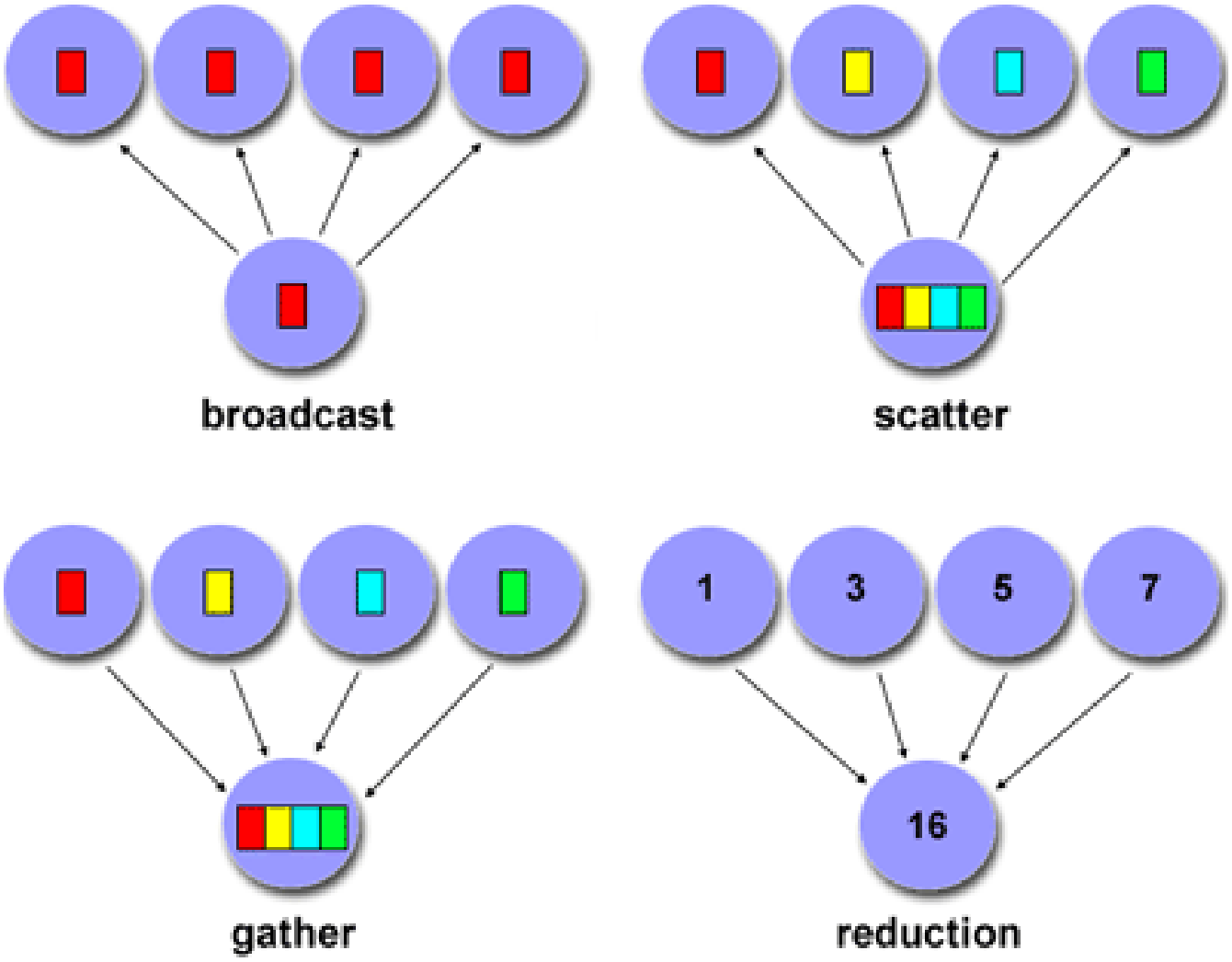
Collective Communication Operations

Collective Communications

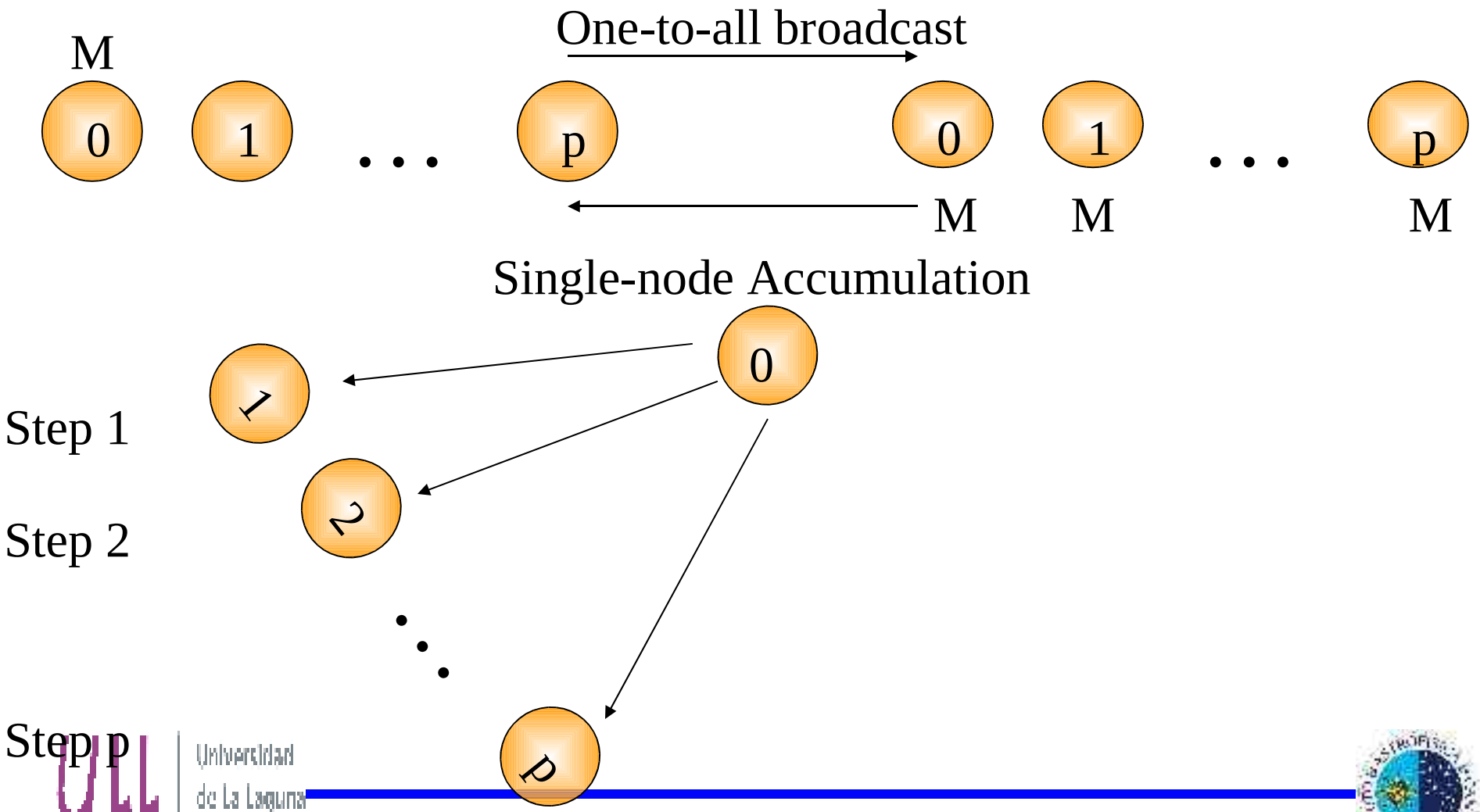
Point-to-point - involves two processes with one process acting as the sender/producer of data, and the other acting as the receiver/consumer.

Collective - involves data sharing between more than two processes, which are often specified as being members in a common group, or collective.

Collective Operations

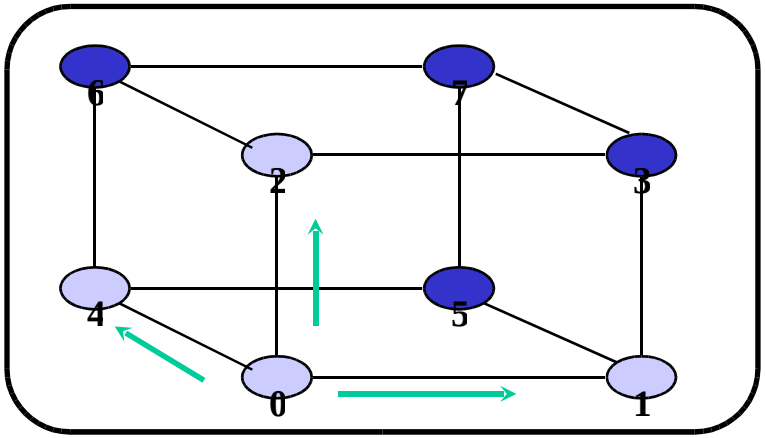


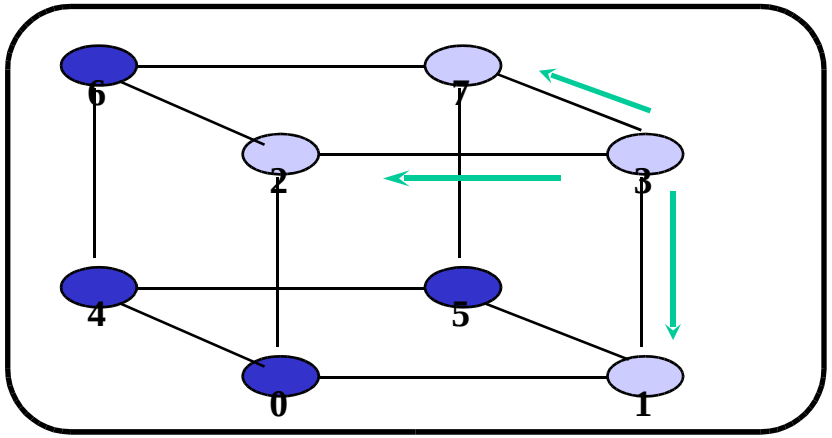
One-to-all broadcast Single-node Accumulation



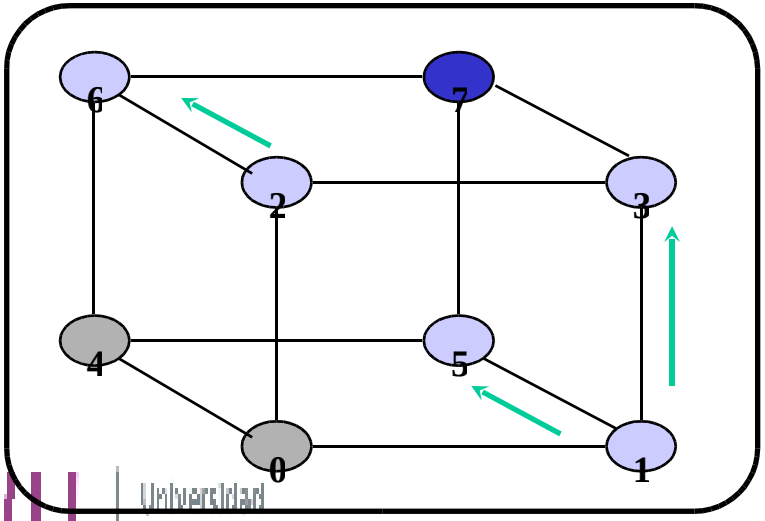
Broadcast on Hypercubes

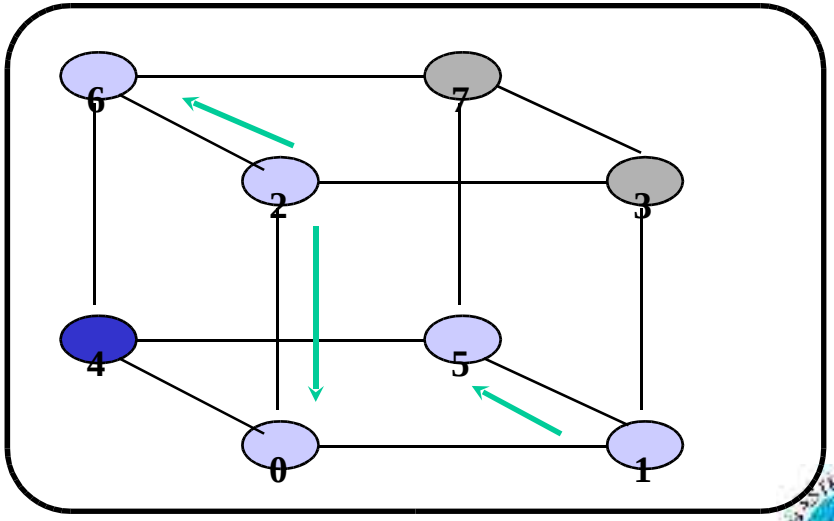
First Step





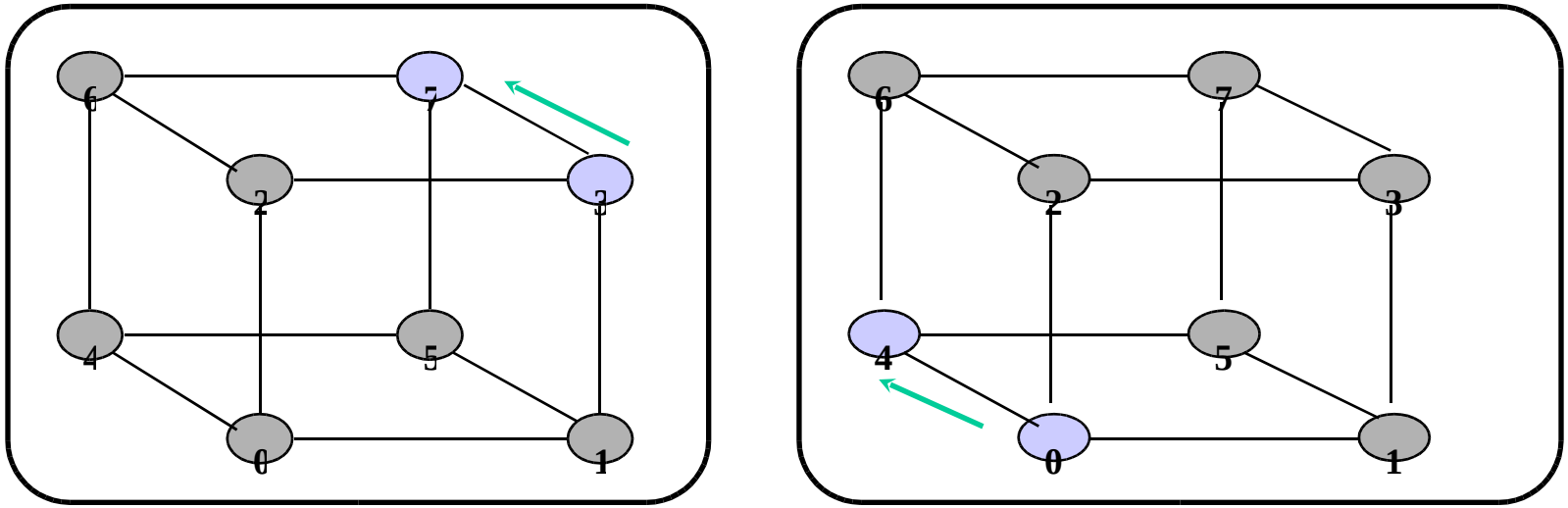
Second Step





Broadcast on Hypercubes

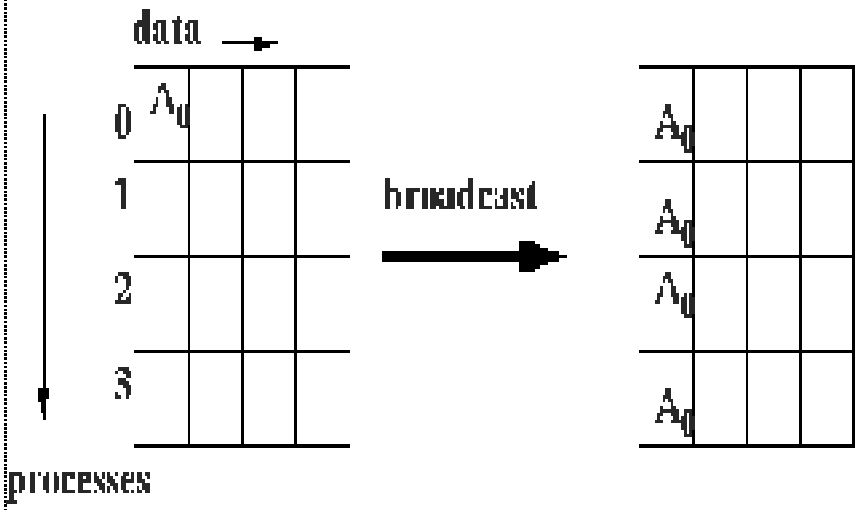
Third Step



MPI Broadcast

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

Broadcasts a message from the process with rank "root" to all other processes of the group

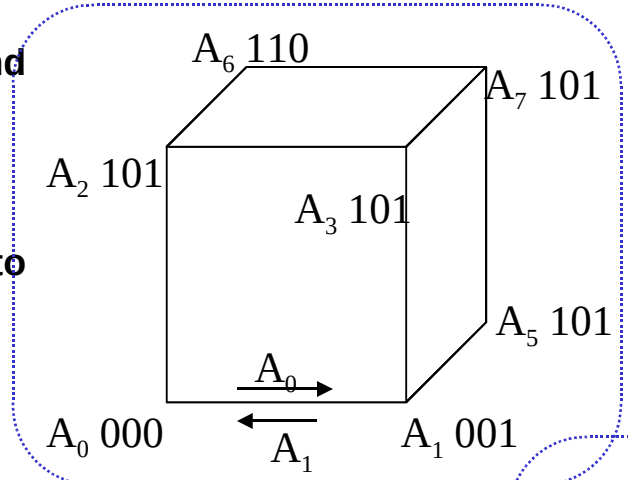


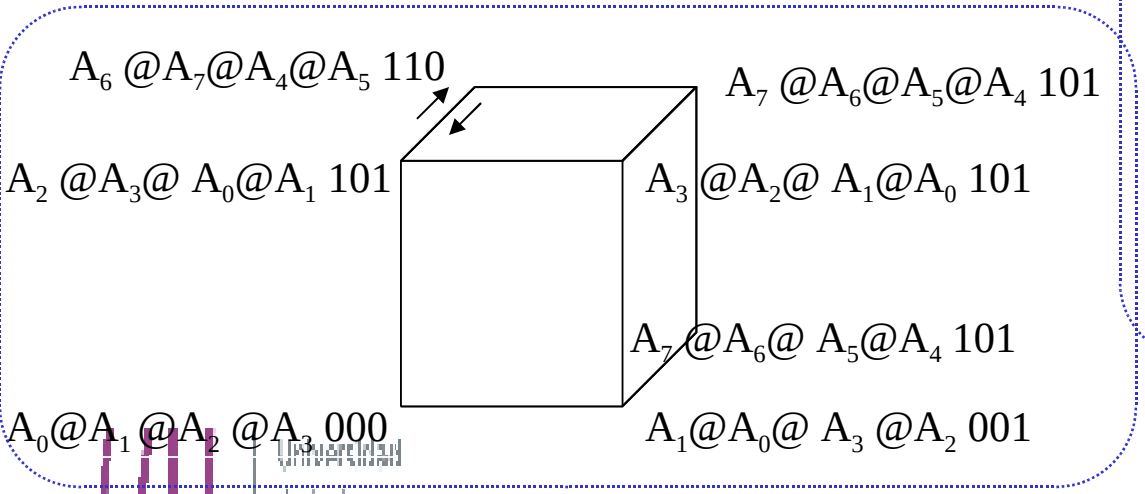
Reduction on Hypercubes

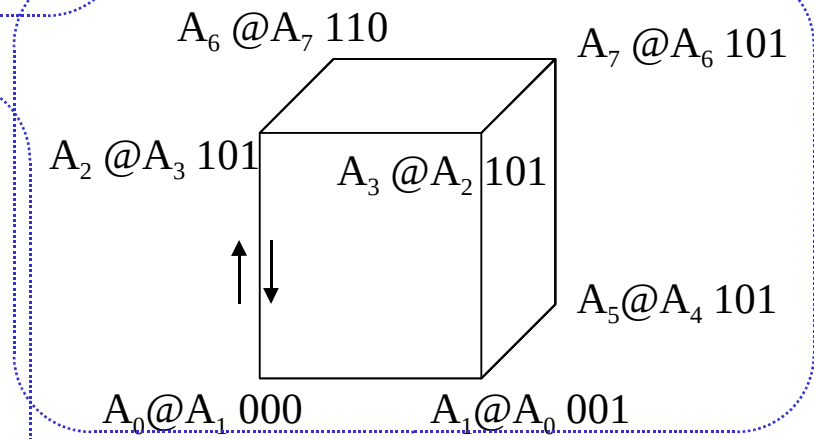
@ conmutative and associative operator

A_i in processor *i*

Every processor has to obtain $A_0@A_1@...@A_{P-1}$







Reductions with MPI

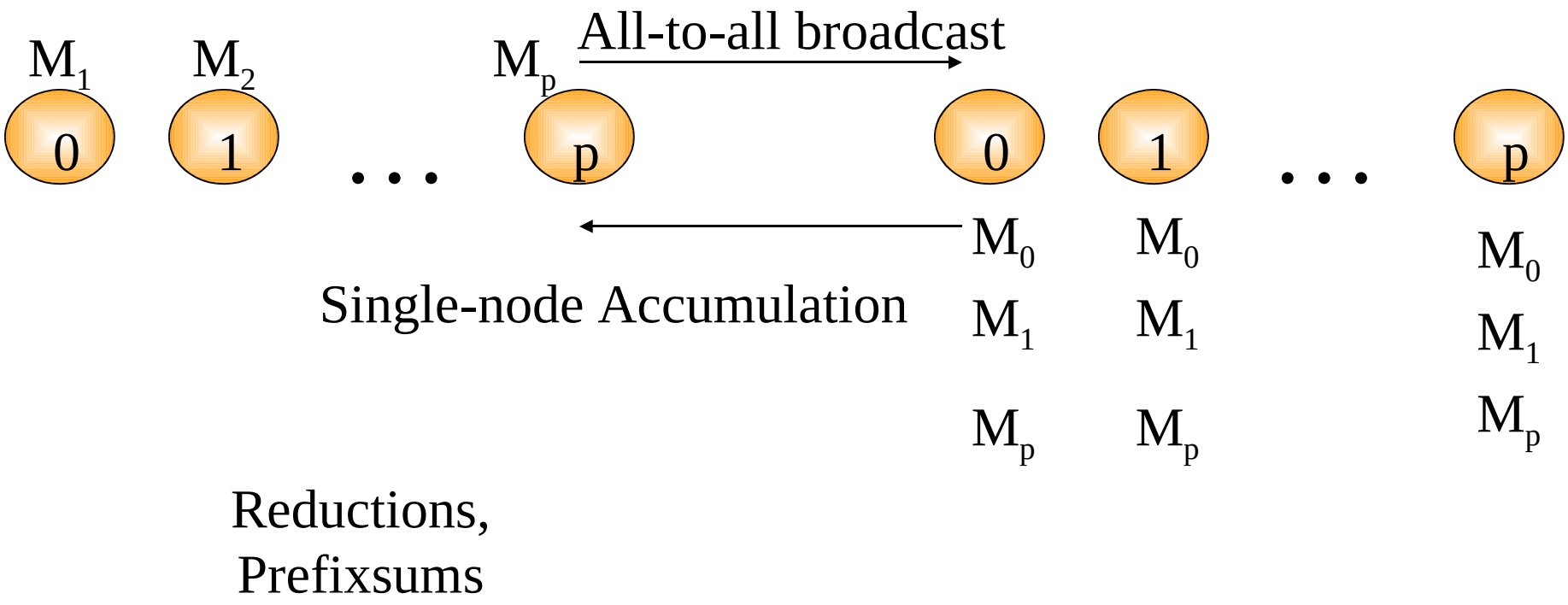
```
int MPI_Reduce(
    void         *sendbuf;
    void         *recvbuf;
    int           count;
    MPI_Datatype  datatype;
    MPI_Op        op;
    int           root;
    MPI_Comm      comm;
);
```

Reduces values on all processes to a single value processes

```
int MPI_Allreduce(
    void         *sendbuf;
    void         *recvbuf;
    int           count;
    MPI_Datatype  datatype;
    MPI_Op        op;
    MPI_Comm      comm;
);
```

Combines values form all processes and distributes the result back to all

All-To-All Broadcast
Multinode Accumulation

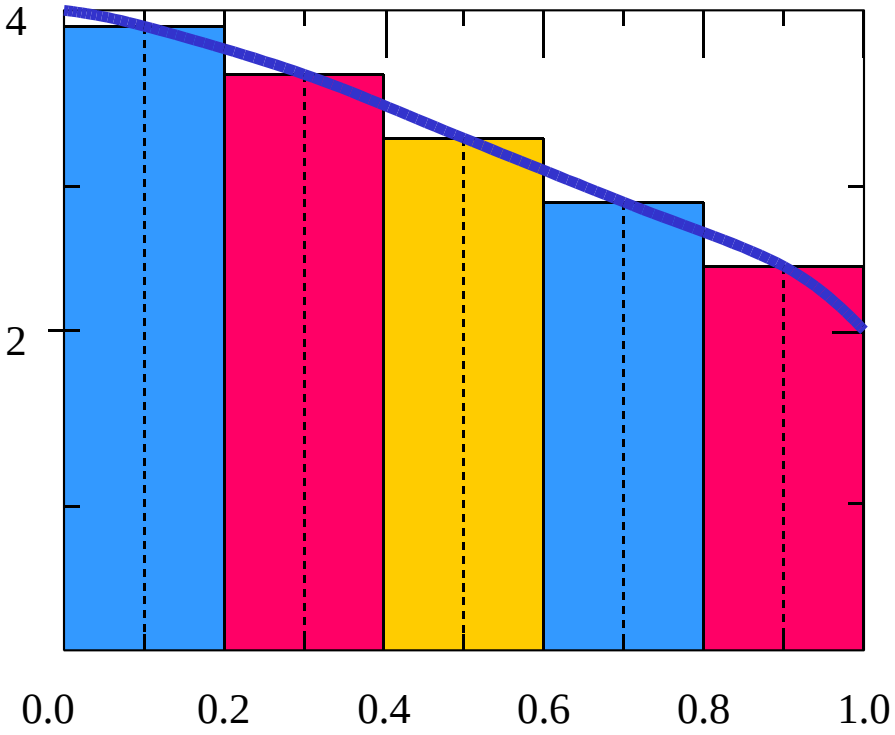


MPI Collective Operations

MPI Operator	Operation
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	max value and location
MPI_MINLOC	min value and location

Computing π: Parallel

$$\pi = \int_0^1 \frac{4}{(1+x^2)} dx$$




```
MPI_Bcast(&n, 1, MPI_INT, 0,
                    MPI_COMM_WORLD);

h   = 1.0 / (double) n;
mypi = 0.0;
for (i = name; i < n; i += numprocs) {
    x = h * (i + 0.5) *h;
    mypi+= f(x);
}
mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,
          MPI_SUM, 0, MPI_COMM_WORLD);
```

ex5/readme.txt

exmatriz/readme.txt

Granularity

Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

Coarse: relatively large amounts of computational work are done between communication events

Fine: relatively small amounts of computational work are done between communication events

Granularity

Fine-grain Parallelism:

Relatively small amounts of computational work are done between communication events

Low computation to communication ratio

Facilitates load balancing

Implies high communication overhead and less opportunity for performance enhancement

If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

Coarse-grain Parallelism:

Relatively large amounts of computational work are done between communication/synchronization events

High computation to communication ratio

Implies more opportunity for performance increase

Harder to load balance efficiently

Granularity

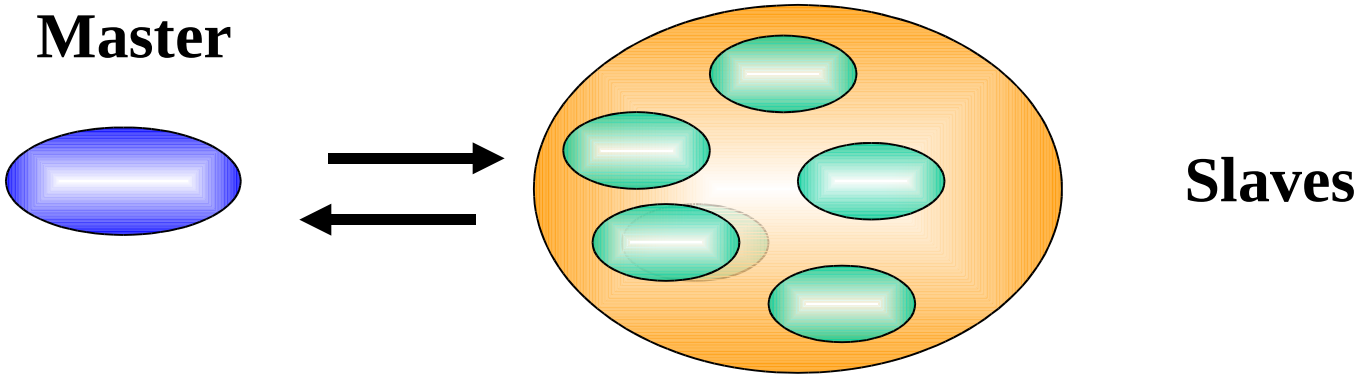
Which is Best?

The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.

In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.

Fine-grain parallelism can help reduce overheads due to load imbalance.

The Master Slave Paradigm



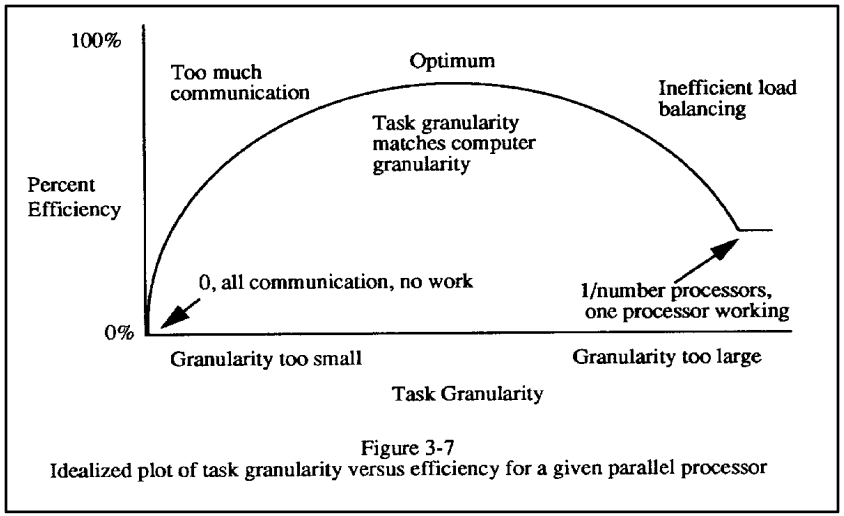
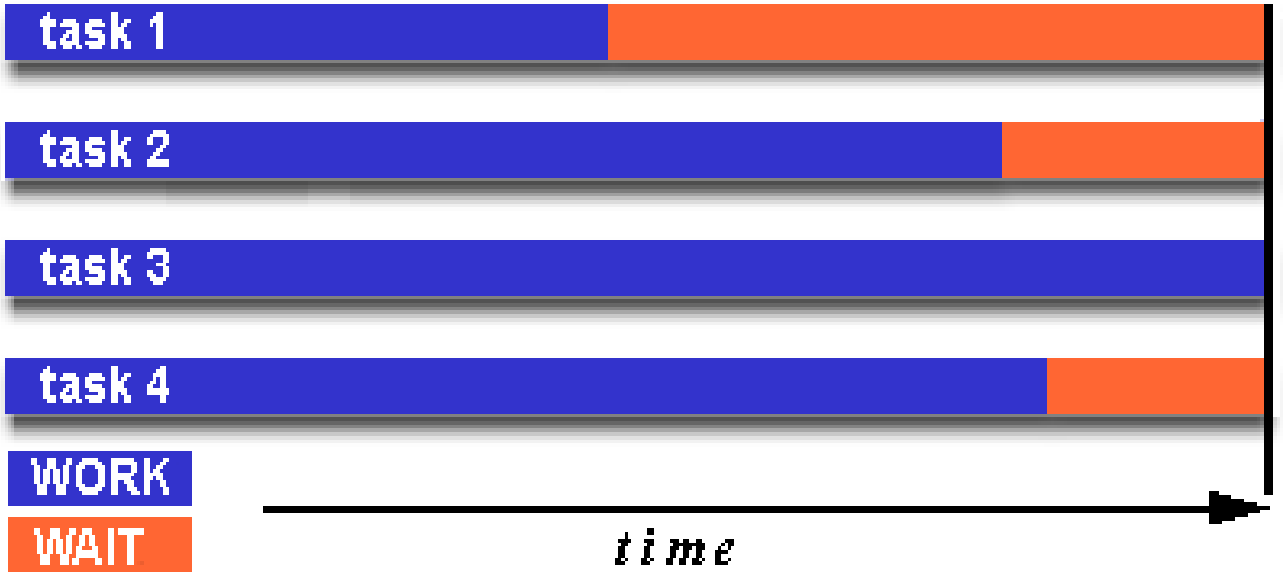


Figure 3-7
Idealized plot of task granularity versus efficiency for a given parallel processor

Load Balance



Load Balance

Equally partition the work each task receives

For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.

For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.

If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

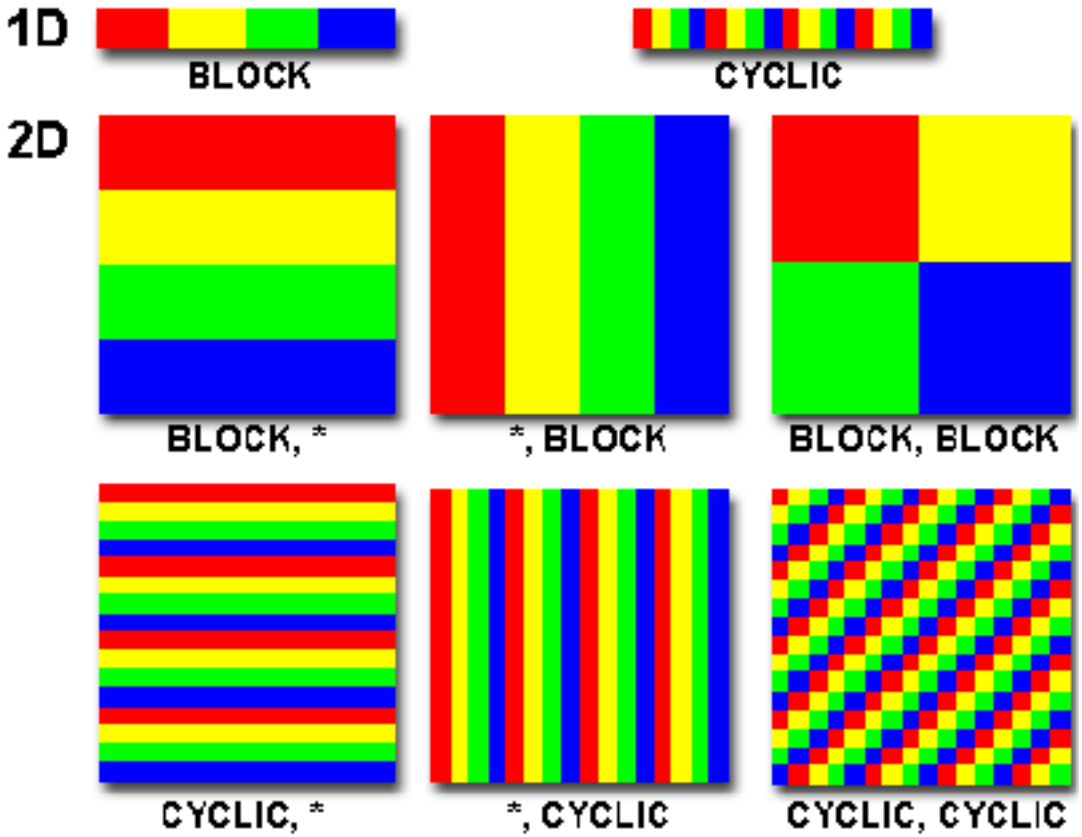
Use dynamic work assignment

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:

When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.

It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Data Partitioning



Hybrid MPI/OpenMP Programming

The Hybrid OpenMP+MPI Programming Model

Combined use of MPI and OpenMP to write parallel programas

Make calls to the MPI run time library

Can be compiled by an OpenMP compiler that inserts calls to a thread library

The model is able to express two levels of parallelism in a program

The Hybrid OpenMP+MPI Programming Model

```
#include <omp.h>
#include <mpi.h>
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs;
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h   = 1.0 / (double) n;
mypi = 0.0;
#pragma omp parallel for reduction(+:mypi) private(x, i)
for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        mypi += f(x);
   }
mypi = h * mypi;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

MPI Related Links

http://www-unix.mcs.anl.gov/mpi/

nereida.deioc.ull.es

Tutorials used in this seminar

- http://www.pdc.kth.se/Talks/MPI/Basic/less.html
- www.llnl.gov/computing/tutorials/parallel_comp

Introducción a la computación de altas prestaciones

¡Gracias!

Francisco Almeida y Francisco de Sande

Departamento de Estadística, I.O. y Computación

Universidad de La Laguna

La Laguna, 12 de febrero de 2004

ULL Universidad de La Laguna