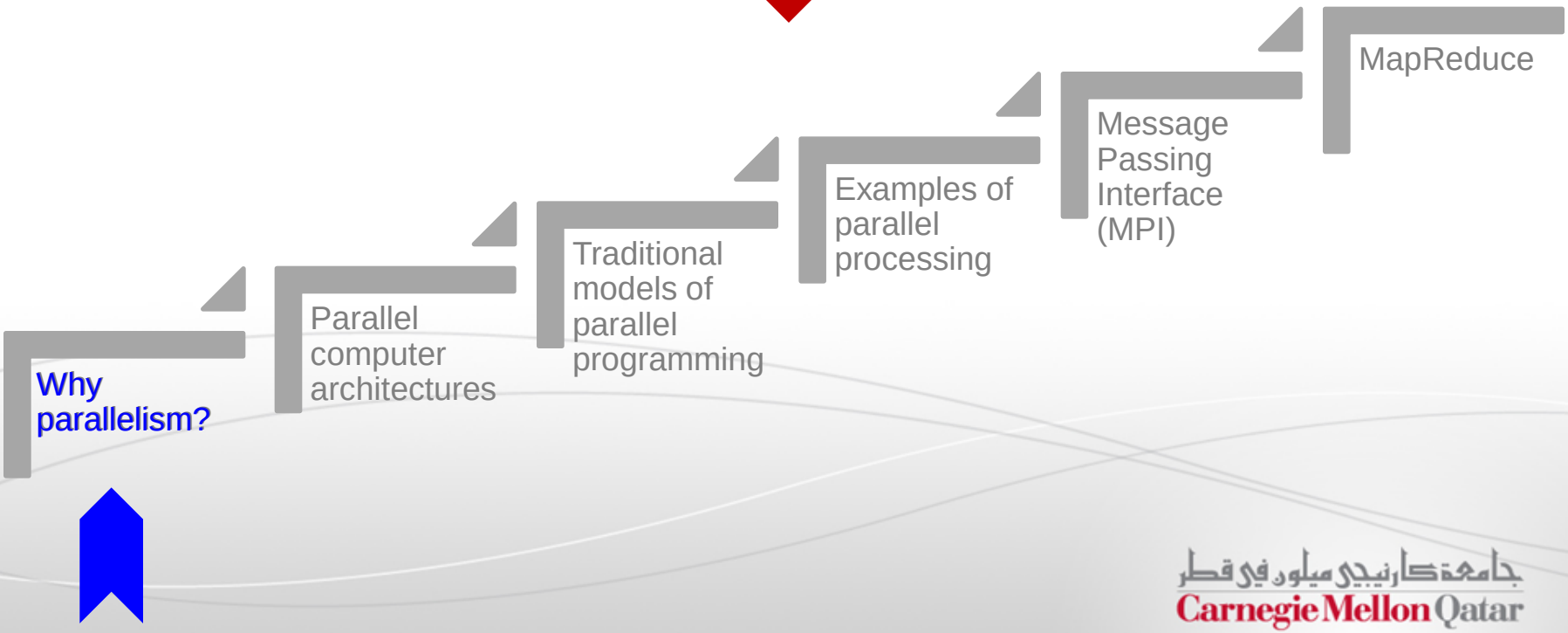
Distributed Systems
CS 15-440/640

Programming Models

Borrowed and adapted from our good friends at
CMU-Doha, Qatar
Majd F. Sakr, Mohammad Hammoud andVinay Kolar

Objectives

Discussion on Programming Models



Amdahl's Law

- We parallelize our programs in order to run them faster
- How much faster will a parallel program run?
  - Suppose that the sequential execution of a program takes T_1 time units and the parallel execution on p processors takes T_p time units
  - Suppose that out of the entire execution of the program, s fraction of it is not parallelizable while $1-s$ fraction is parallelizable
  - Then the speedup (***Amdahl's formula***):

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$

Amdahl's Law: An Example

- Suppose that 80% of you program can be parallelized and that you use 4 processors to run your parallel version of the program

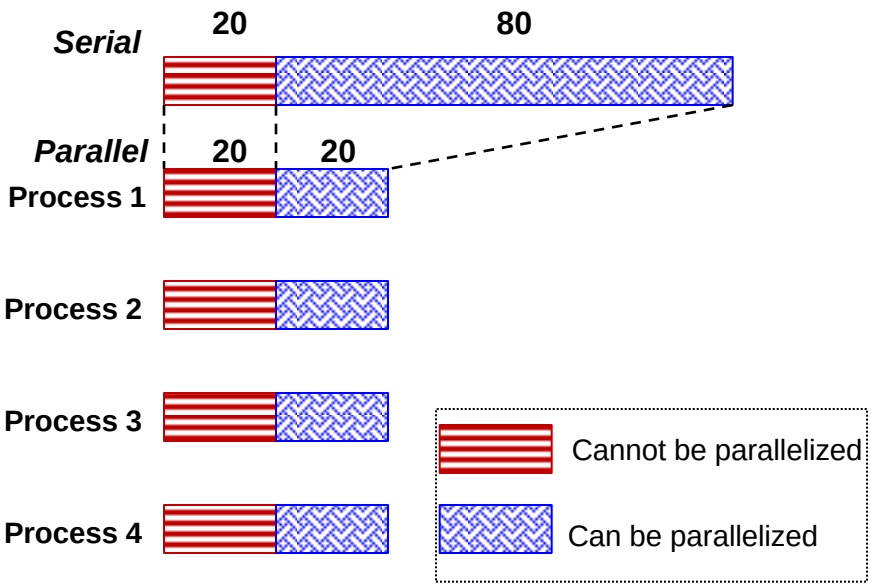
- The speedup you can get according to Amdahl is:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

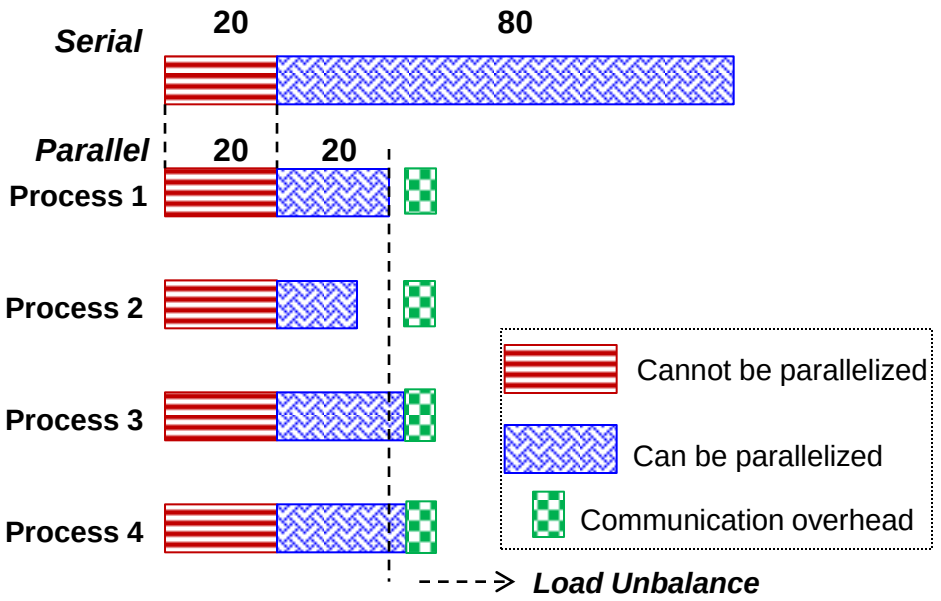
- Although you use 4 processors you cannot get a speedup more than 2.5 times (or 40% of the serial running time)

Real Vs. Actual Cases

- Amdahl's argument is too simplified to be applied to real cases
- When we run a parallel program, there are a communication overhead and a workload imbalance among processes in general



1. Parallel Speed-up: An Ideal Case



2. Parallel Speed-up: An Actual Case

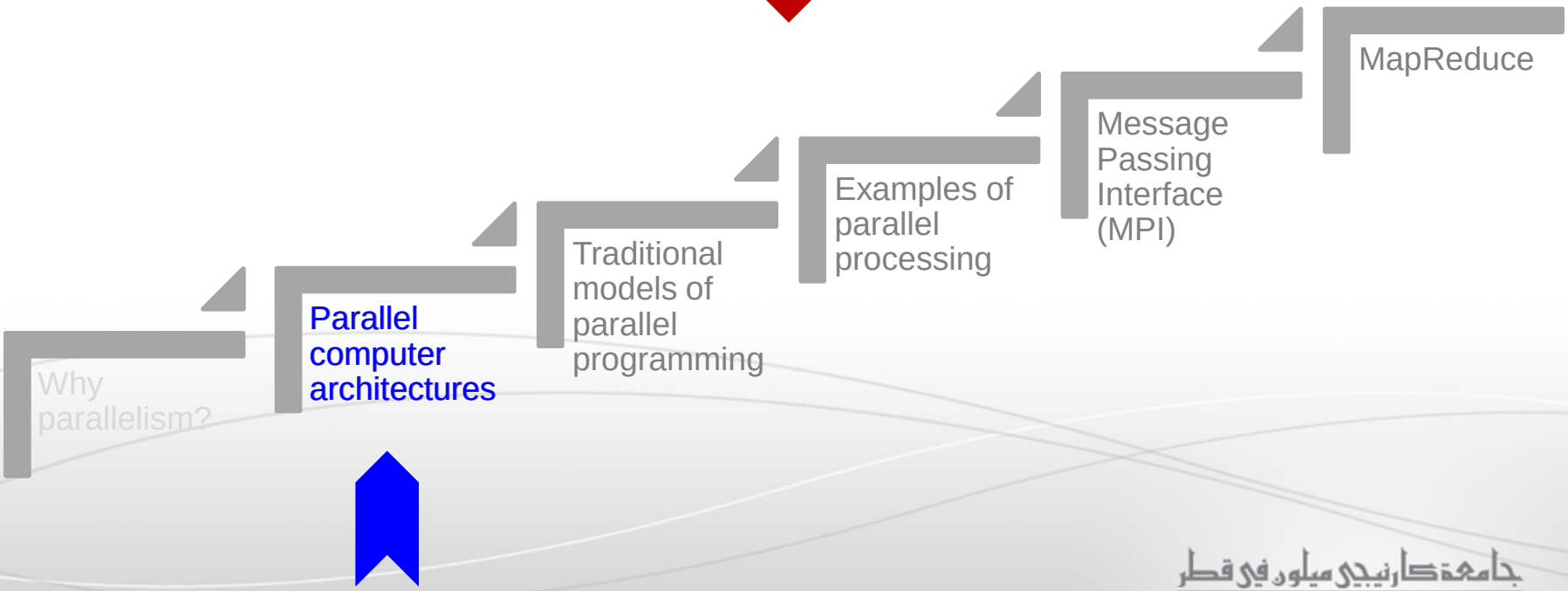
Guidelines

- In order to efficiently benefit from parallelization, we ought to follow these guidelines:

1. Maximize the fraction of our program that can be parallelized
2. Balance the workload of parallel processes
3. Minimize the time spent for communication

Objectives

Discussion on Programming Models



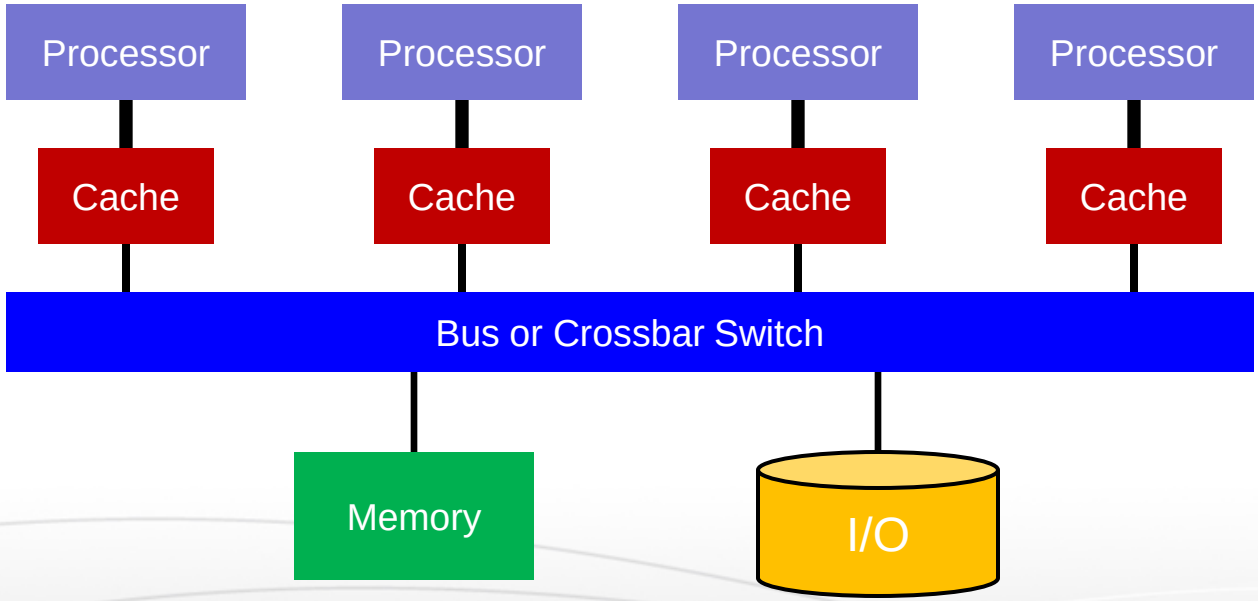
Parallel Computer Architectures

- We can categorize the architecture of parallel computers in terms of two aspects:
  - Whether the memory is physically centralized or distributed
  - Whether or not the address space is shared

Memory		Address Space	
		Shared	**Individual**
	Centralized	UMA – SMP (Symmetric Multiprocessor)	N/A
	Distributed	NUMA (Non-Uniform Memory Access)	MPP (Massively Parallel Processors)

Symmetric Multiprocessor

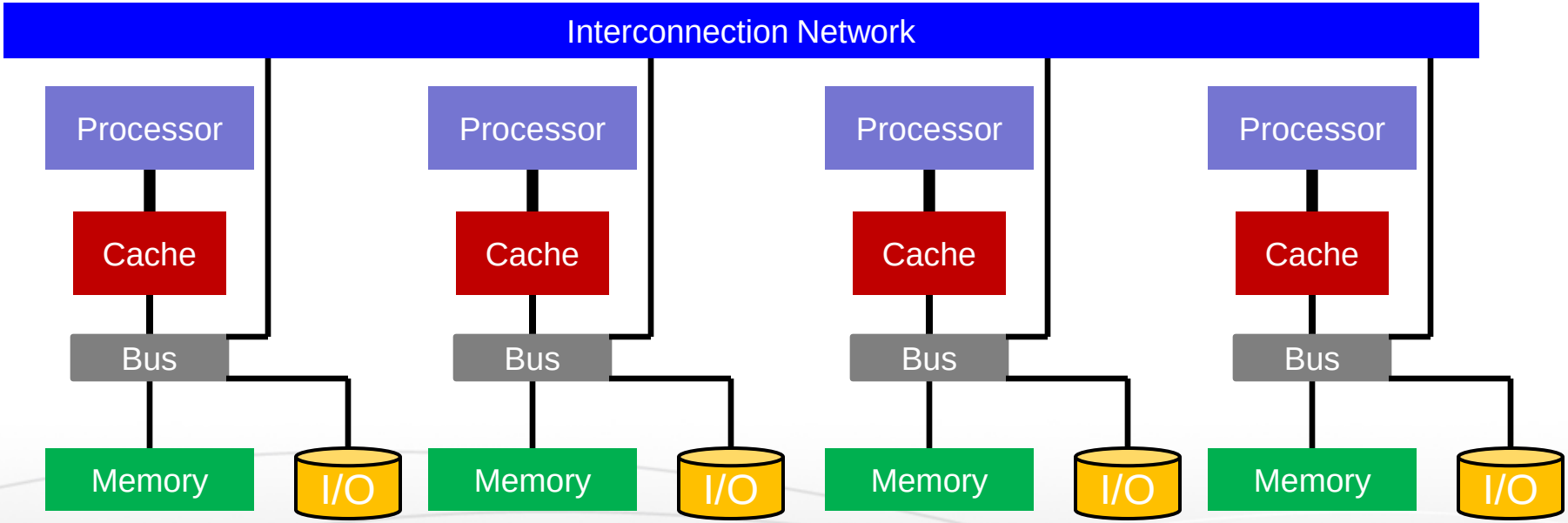
- Symmetric Multiprocessor (SMP) architecture uses shared system resources that can be accessed equally from all processors



- A single OS controls the SMP machine and it schedules processes and threads on processors so that the load is balanced

Massively Parallel Processors

- Massively Parallel Processors (MPP) architecture consists of nodes with each having its own processor, memory and I/O subsystem



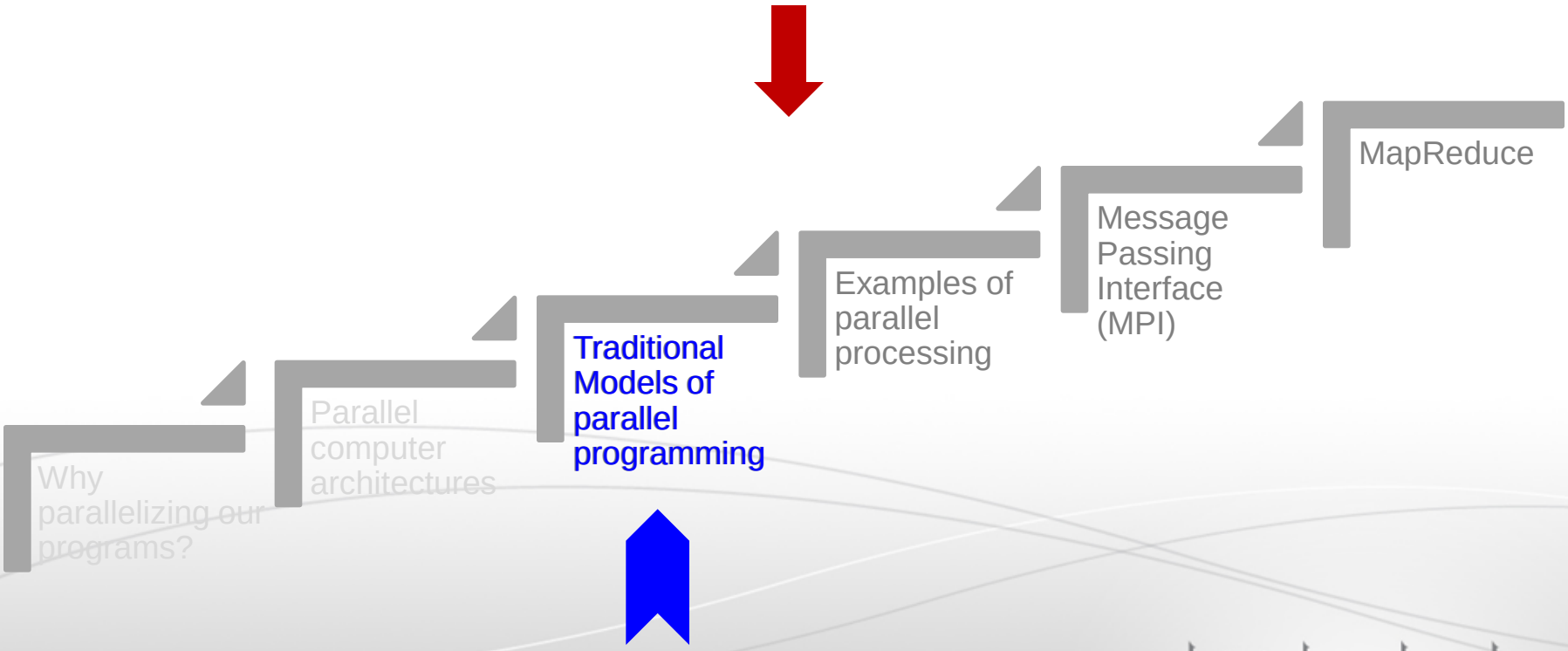
- An independent OS runs at each node

Non-Uniform Memory Access

- Non-Uniform Memory Access (NUMA) architecture machines are built on a similar hardware model as MPP
- NUMA typically provides a shared address space to applications using a hardware/software directory-based coherence protocol
- The memory latency varies according to whether you access memory directly (local) or through the interconnect (remote). Thus the name non-uniform memory access
- As in an SMP machine, a single OS controls the whole system

Objectives

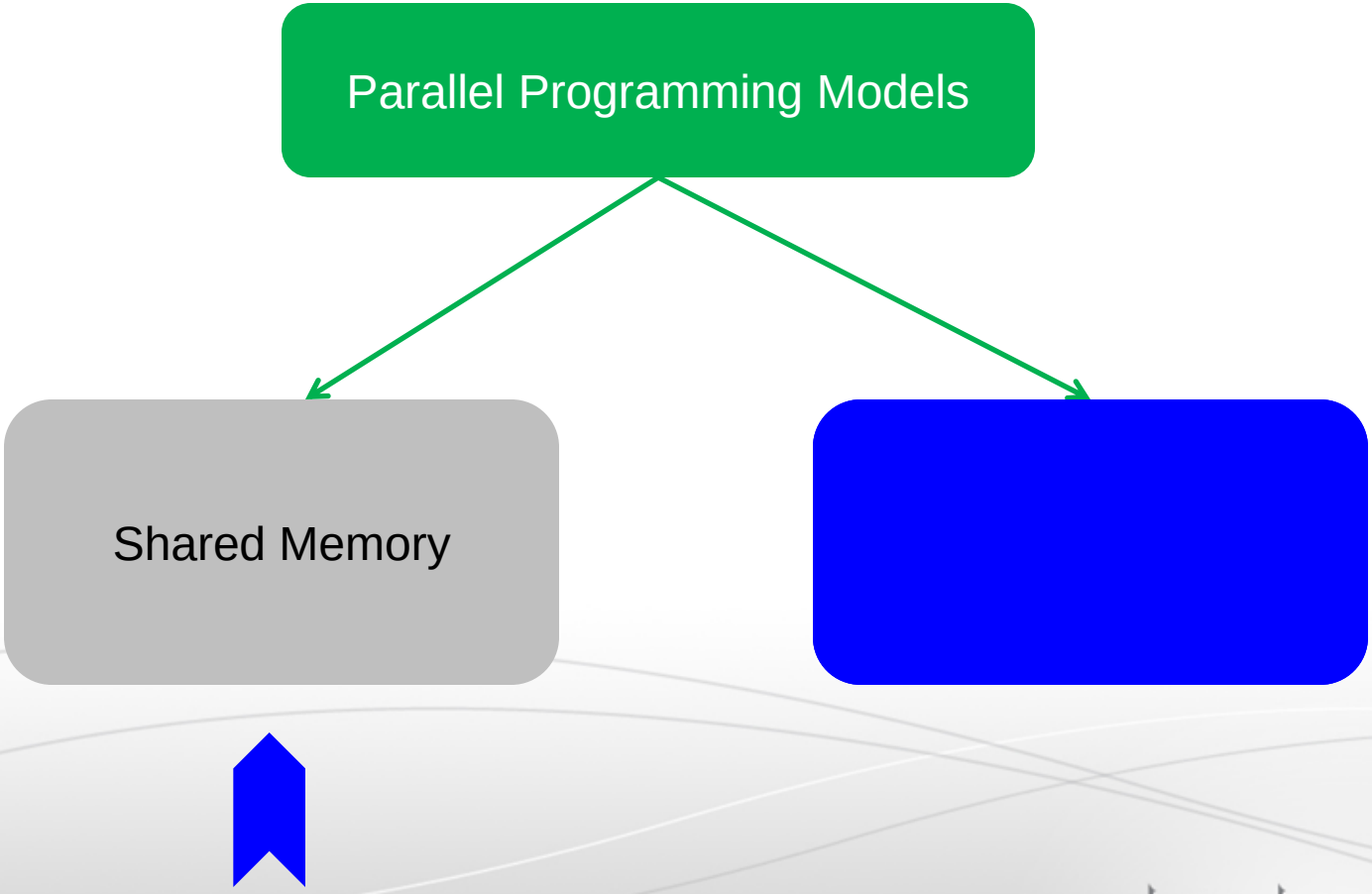
Discussion on Programming Models



Models of Parallel Programming

- What is a parallel programming model?
  - A programming model is an abstraction provided by the hardware to programmers
  - It determines how easily programmers can specify their algorithms into parallel unit of computations (i.e., tasks) that the hardware understands
  - It determines how efficiently parallel tasks can be executed on the hardware
- Main Goal: utilize all the processors of the underlying architecture (e.g., SMP, MPP, NUMA) and minimize the elapsed time of your program

Traditional Parallel Programming Models

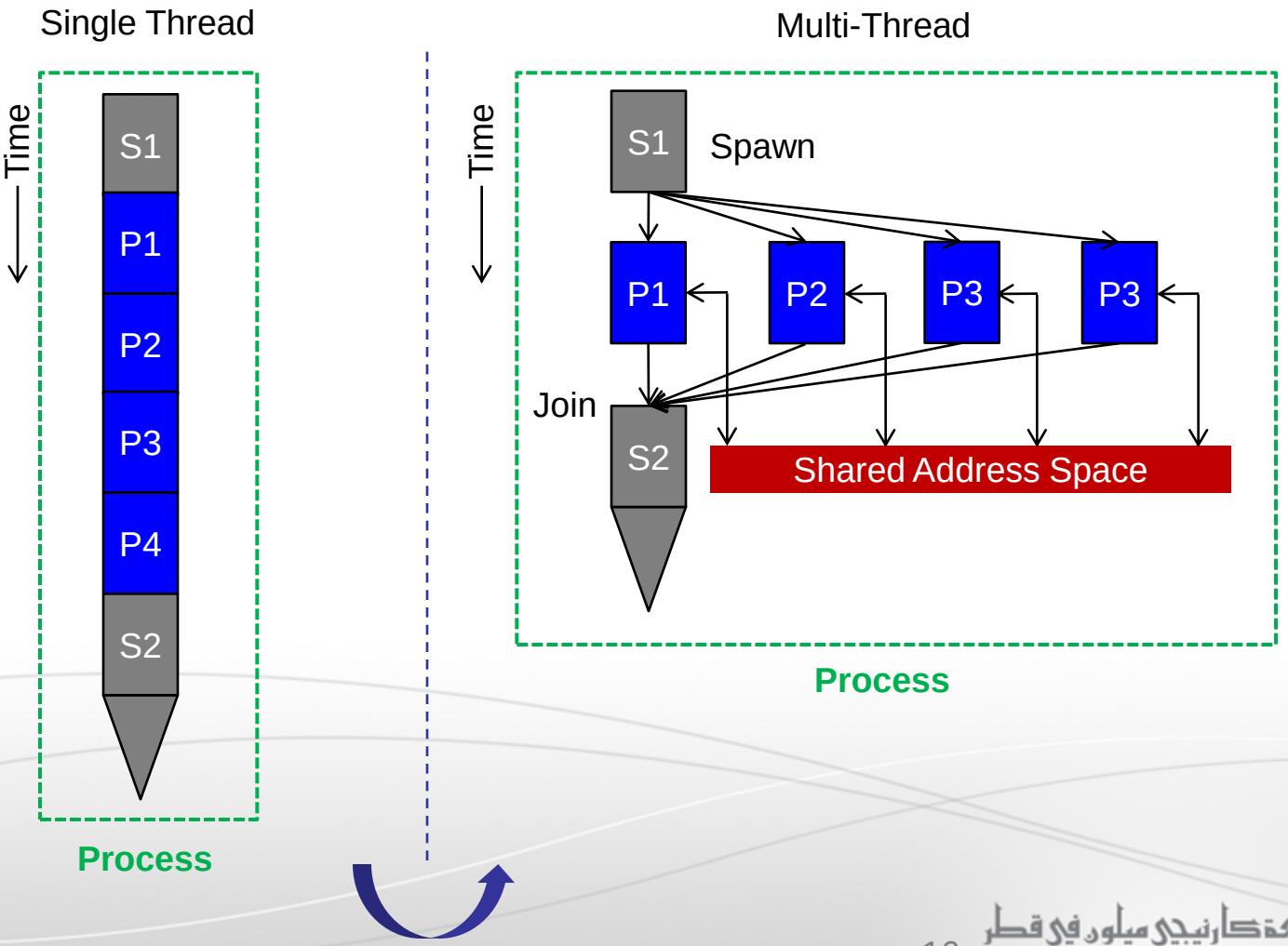


Shared Memory Model

- In the shared memory programming model, the abstraction is that parallel tasks can access any location of the memory
- Parallel tasks can communicate through reading and writing common memory locations
- This is similar to threads from a single process which share a single address space
- Multi-threaded programs (e.g., OpenMP programs) are the best fit with shared memory programming model

Shared Memory Model

S_i = *Serial*
P_j = *Parallel*



Shared Memory Example

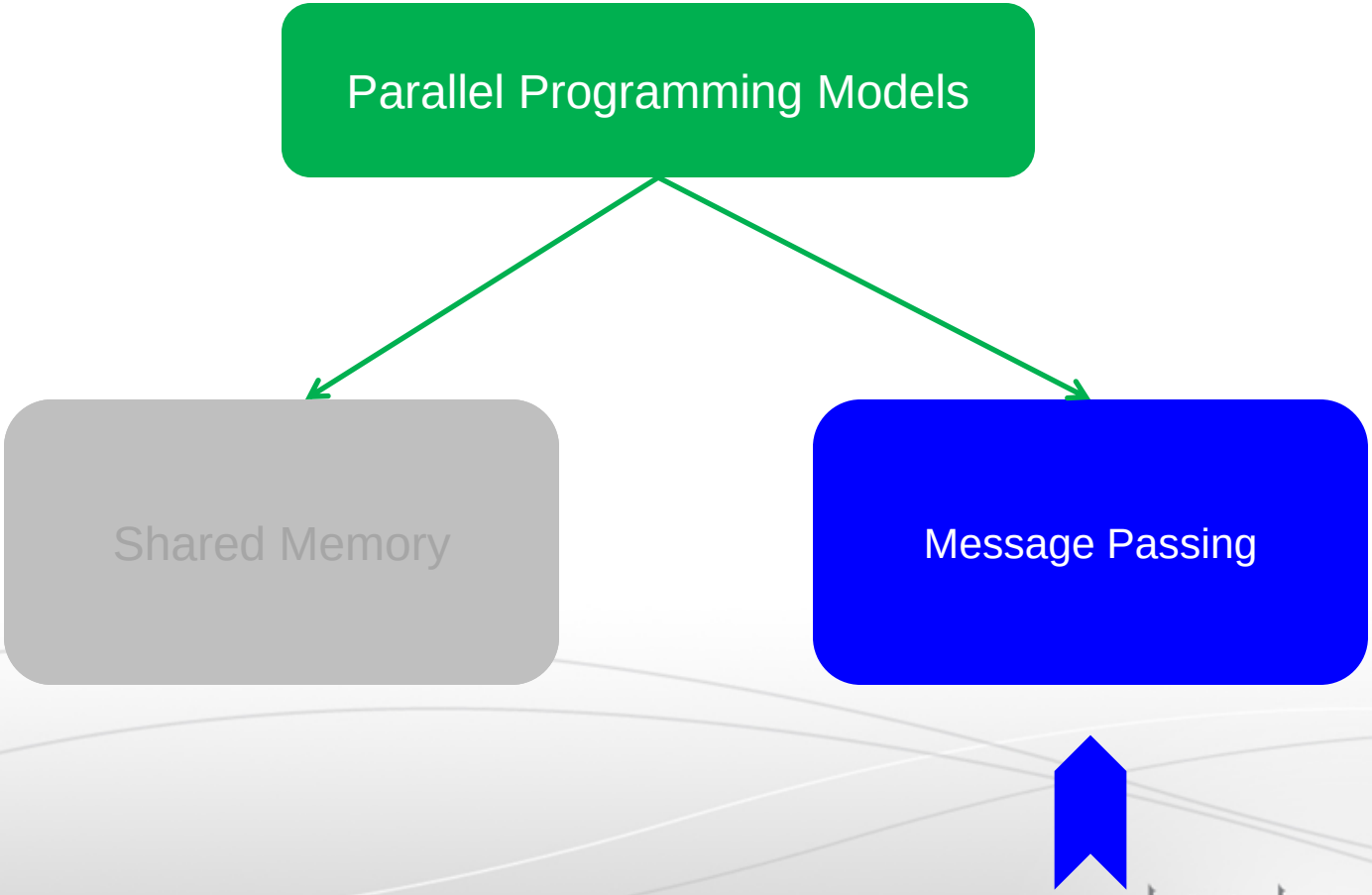
```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

Sequential

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter=4, sum=0;
shared double sum=0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
      sum = sum + a[i];
    unlock(mylock);
  }
barrier;     // necessary

end parallel // kill the child thread
Print sum;
```

Parallel

Traditional Parallel Programming Models

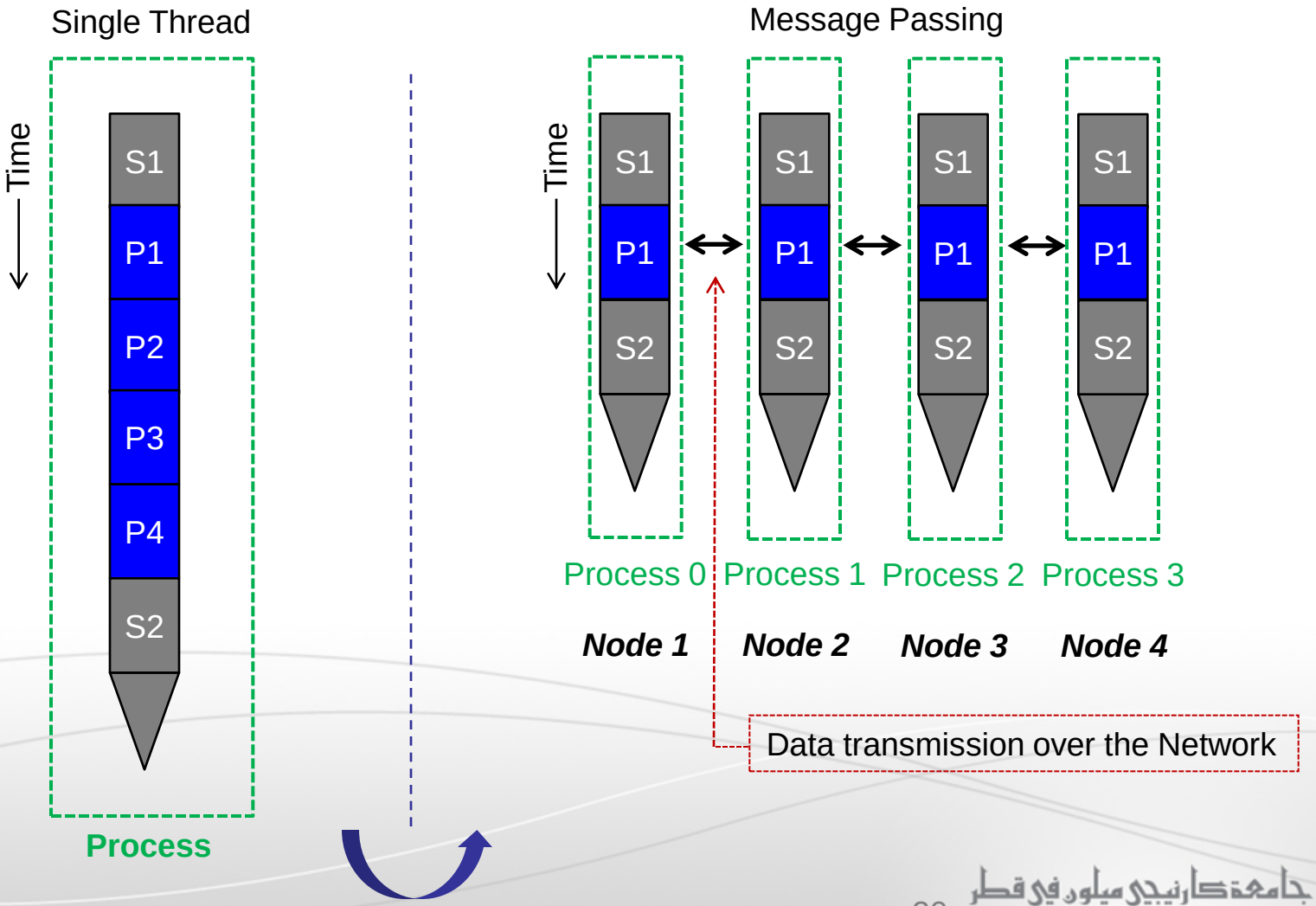


Message Passing Model

- In message passing, parallel tasks have their own local memories
- One task cannot access another task's memory
- Hence, to communicate data they have to rely on explicit messages sent to each other
- This is similar to the abstraction of processes which do not share an address space
- MPI programs are the best fit with message passing programming model

Message Passing Model

S = Serial
P = Parallel



Message Passing Example

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
sum = 0;
for (i=0; i<8; i++)
  if (a[i] > 0)
    sum = sum + a[i];
Print sum;
```

Sequential

```
id = getpid();
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4..7]);
else
  recv_msg (P0, b[4..7], c[4..7]);

for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0)
    local_sum = local_sum + a[i];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}
else
  send_msg (P0, local_sum);
```

Parallel

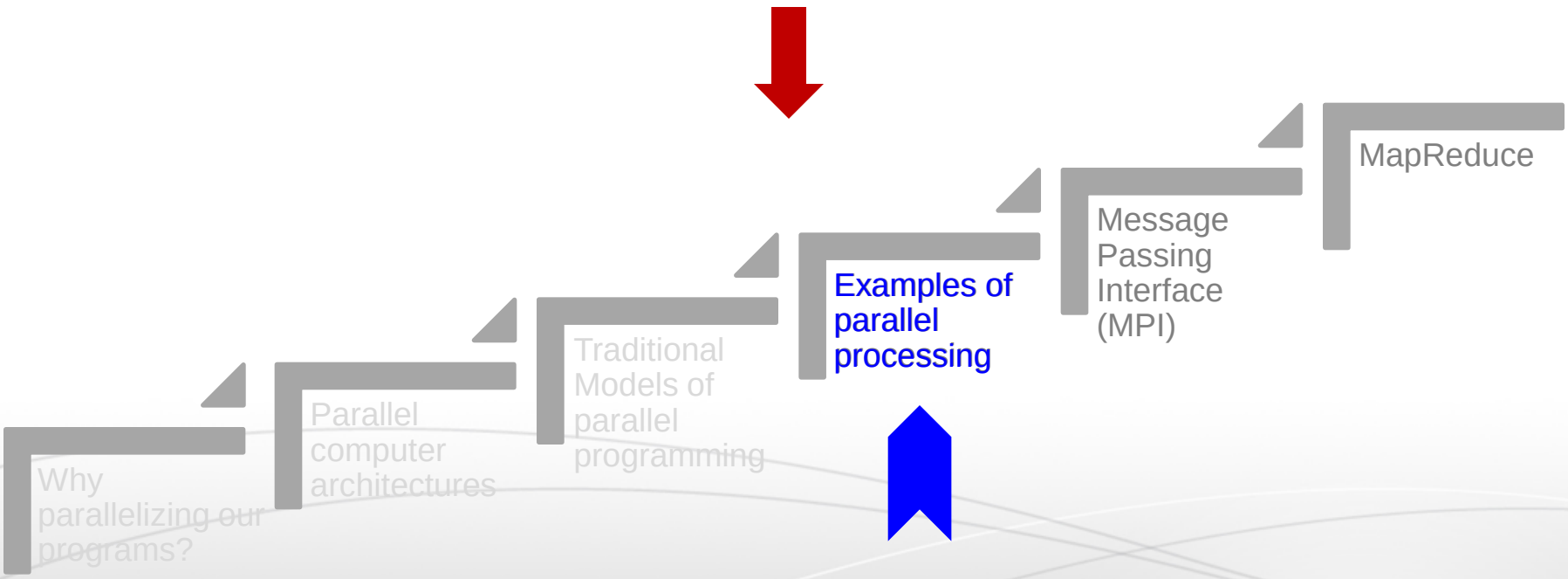
Shared Memory Vs. Message Passing

- Comparison between shared memory and message passing programming models:

Aspect	Shared Memory	Message Passing
Communication	Implicit (via loads/stores)	Explicit Messages
Synchronization	Explicit	Implicit (Via Messages)
Hardware Support	Typically Required	None
Development Effort	Lower	Higher
Tuning Effort	Higher	Lower

Objectives

Discussion on Programming Models



SPMD and MPMD

- When we run multiple processes with message-passing, there are further categorizations regarding how many *different* programs are cooperating in parallel execution

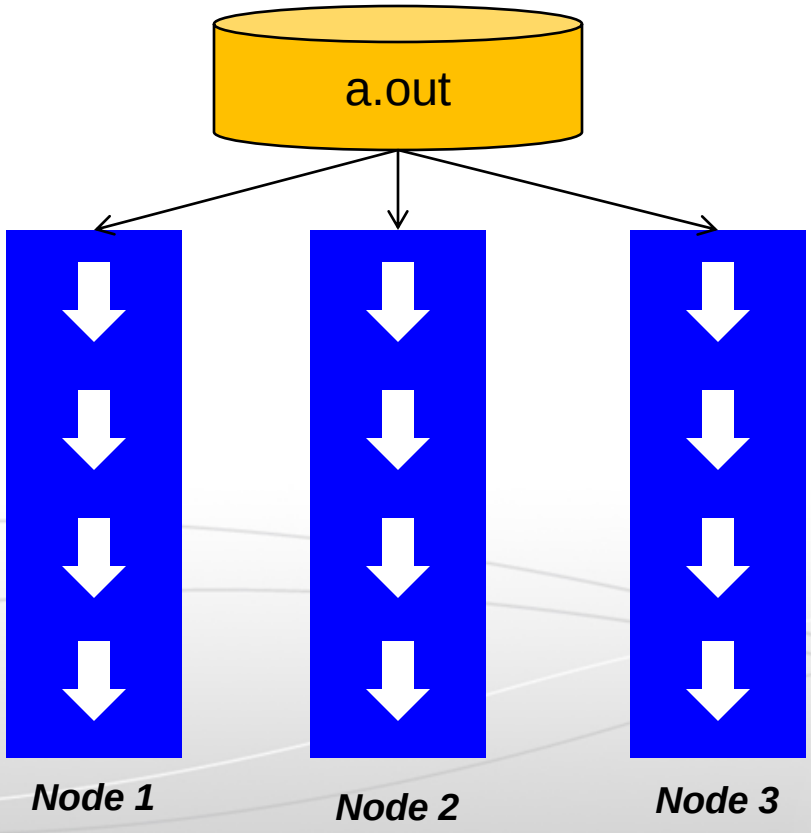
- We distinguish between two models:

  1. Single Program Multiple Data (**SPMD**) model

  2. Multiple Programs Multiple Data (**MPMP**) model

SPMD

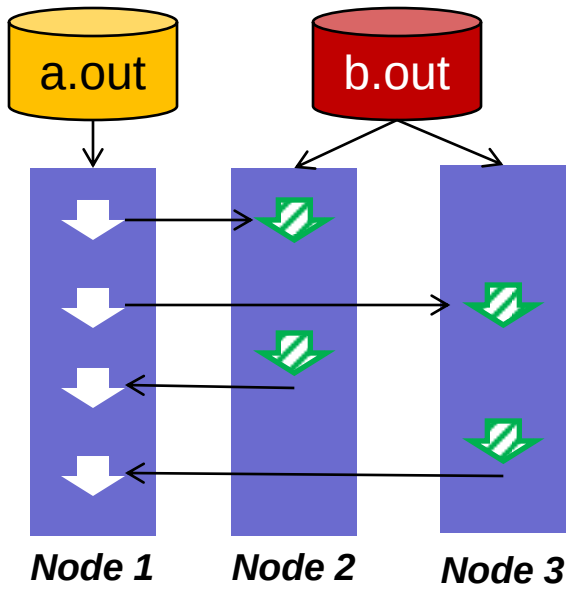
- In the SPMD model, there is only one program and each process uses the same executable working on different sets of data

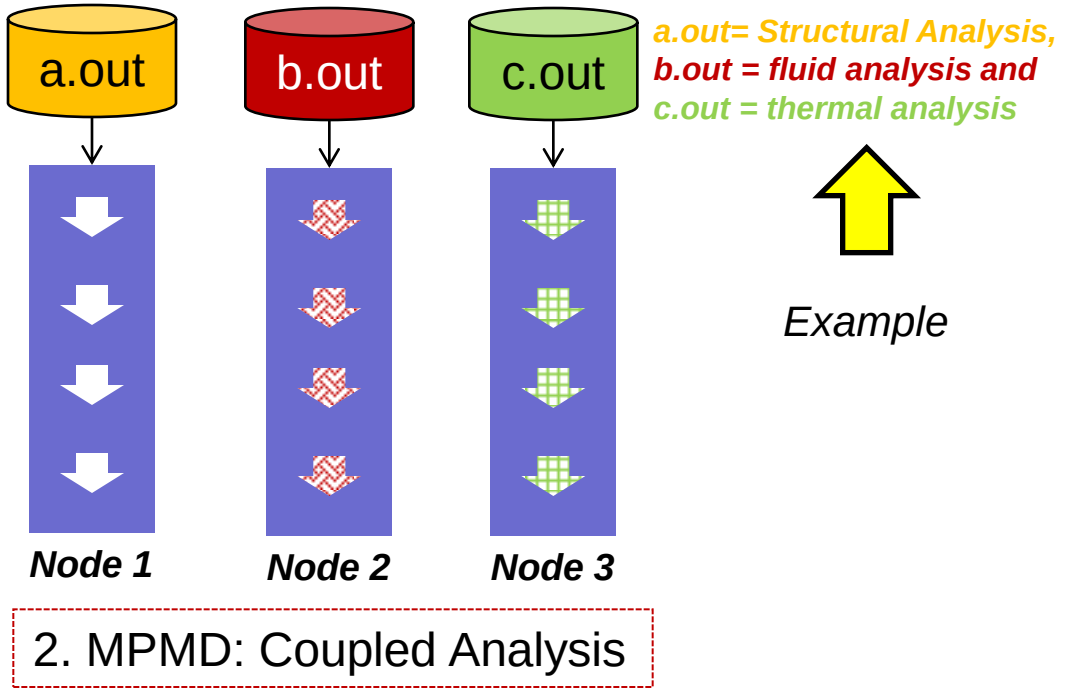


MPMD

- The MPMD model uses different programs for different processes, but the processes collaborate to solve the same problem

- MPMD has two styles, the *master/worker* and the *coupled analysis*



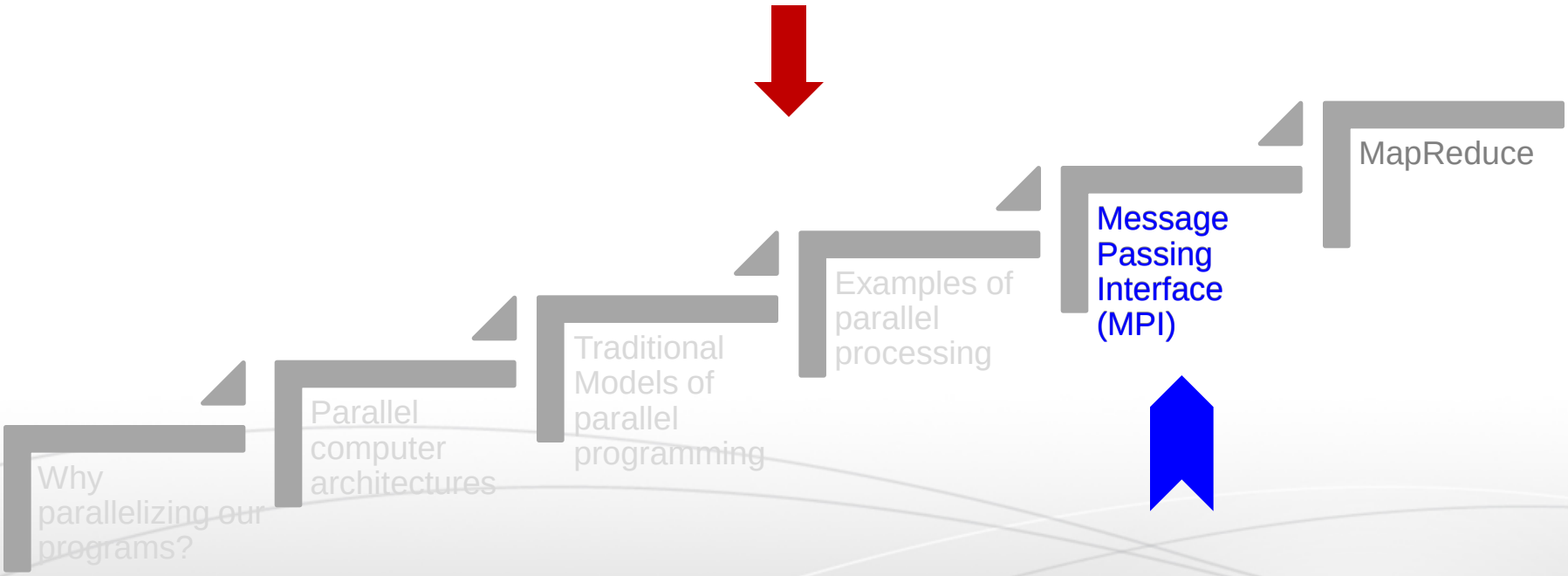


3 Key Points

- To summarize, keep the following 3 points in mind:
  - The purpose of parallelization is to reduce the time spent for computation
  - Ideally, the parallel program is *p* times faster than the sequential program, where *p* is the number of processes involved in the parallel execution, *but this is not always achievable*
  - Message-passing is the tool to consolidate what parallelization has separated. It should not be regarded as the parallelization itself

Objectives

Discussion on Programming Models



Message Passing Interface

- In this part, the following concepts of MPI will be described:
  - Basics
  - Point-to-point communication
  - Collective communication

What is MPI?

- The Message Passing Interface (MPI) is a message passing library standard for writing message passing programs
- The goal of MPI is to establish a *portable*, *efficient*, and *flexible* standard for message passing
- By itself, MPI is NOT a library - but rather the specification of what such a library should be
- MPI is not an IEEE or ISO standard, but has in fact, become the *industry standard* for writing message passing programs on HPC platforms

Reasons for using MPI

Reason	Description
Standardization	MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms
Portability	There is no need to modify your source code when you port your application to a different platform that supports the MPI standard
Performance Opportunities	Vendor implementations should be able to exploit native hardware features to optimize performance
Functionality	Over 115 routines are defined
Availability	A variety of implementations are available, both vendor and public domain

Programming Model

- MPI is an example of a message passing programming model
- MPI is now used on just about any common parallel architecture including MPP, SMP clusters, workstation clusters and heterogeneous networks
- With MPI the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Communicators and Groups

- MPI uses objects called *communicators and groups* to define which collection of processes may communicate with each other to solve a certain problem

- Most MPI routines require you to specify a communicator as an argument

- The communicator **MPI_COMM_WORLD** is often used in calling communication subroutines

- MPI_COMM_WORLD is the predefined communicator that includes *all* of your MPI processes

Ranks

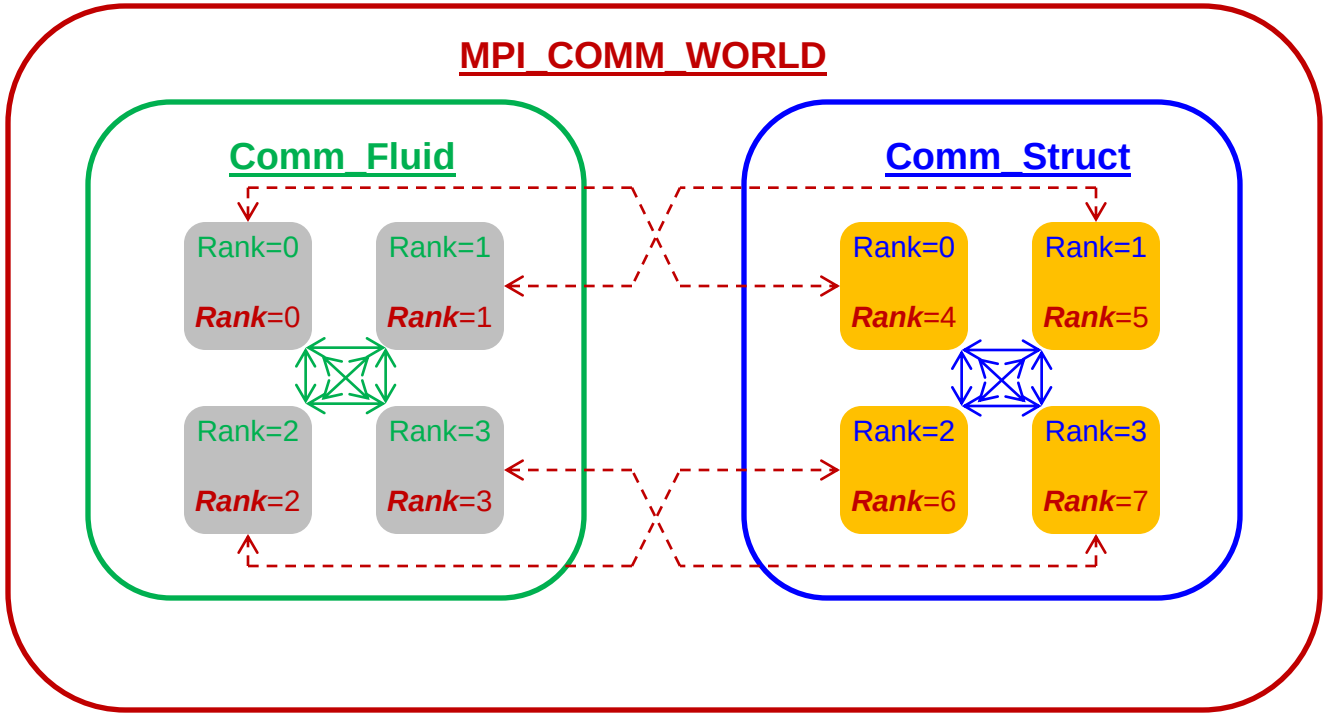
- Within a communicator, every process has its own unique, integer identifier referred to as *rank*, assigned by the system when the process initializes
- A rank is sometimes called a *task ID*. Ranks are contiguous and begin at *zero*
- Ranks are used by the programmer to specify the source and destination of messages
- Ranks are often also used conditionally by the application to control program execution (e.g., *if rank=0 do this / if rank=1 do that*)

Multiple Communicators

- It is possible that a problem consists of several sub-problems where each can be solved concurrently
- This type of application is typically found in the category of MPMD coupled analysis
- We can create a new communicator for each sub-problem as a subset of an existing communicator
- MPI allows you to achieve that by using **MPI_COMM_SPLIT**

Example of Multiple Communicators

- Consider a problem with a fluid dynamics part and a structural analysis part, where each part can be computed in parallel



- ✓ Ranks within MPI_COMM_WORLD are printed in red
- ✓ Ranks within Comm_Fluid are printed with green
- ✓ Ranks within Comm_Struct are printed with blue