Cloud Computing
CS 15-319

Programming Models- Part I

Lecture 4, Jan 25, 2012

Majd F. Sakr and Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

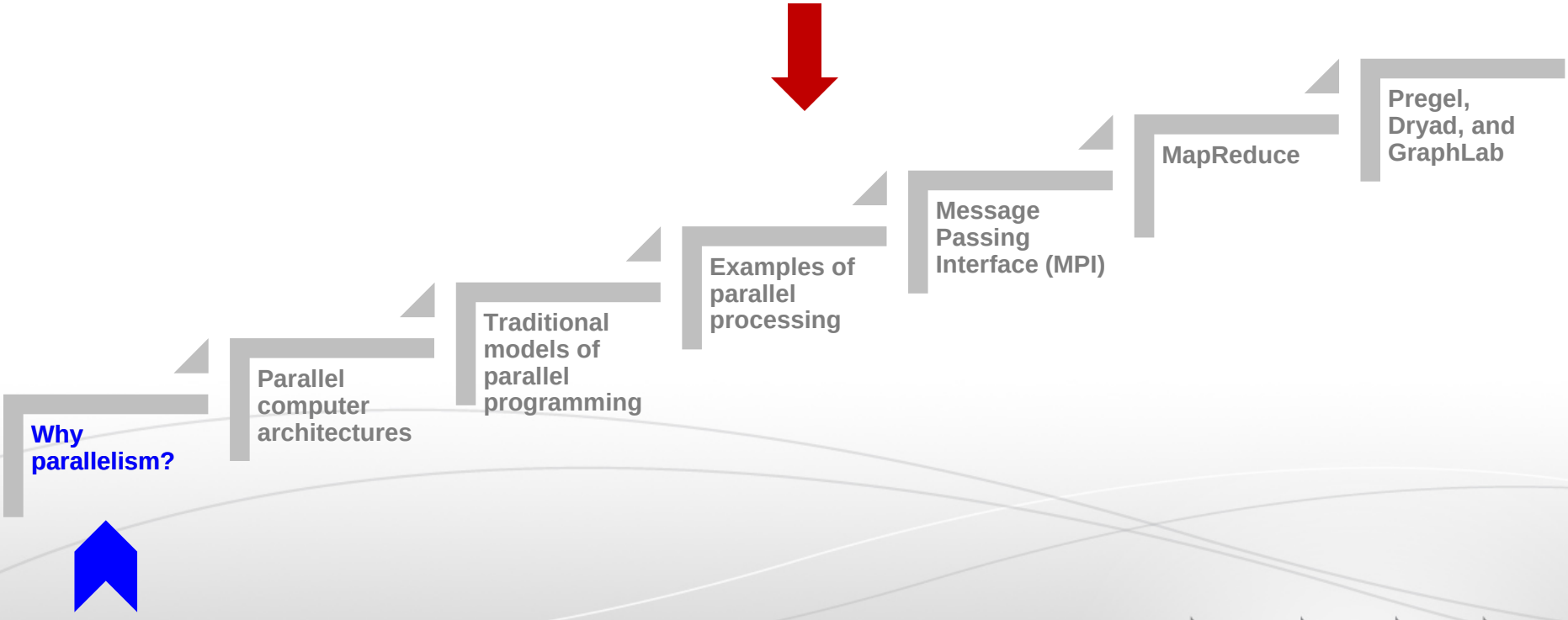
Today…

- Last 3 sessions
  - Administrivia and Introduction to Cloud Computing
  - Introduction to Cloud Computing and Cloud Software Stack
  - Course Project and Amazon AWS
- Today's session
  - Programming Models – *Part I*
- Announcement:
  - Project update is due today

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Objectives

Discussion on Programming Models



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

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***):

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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

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:

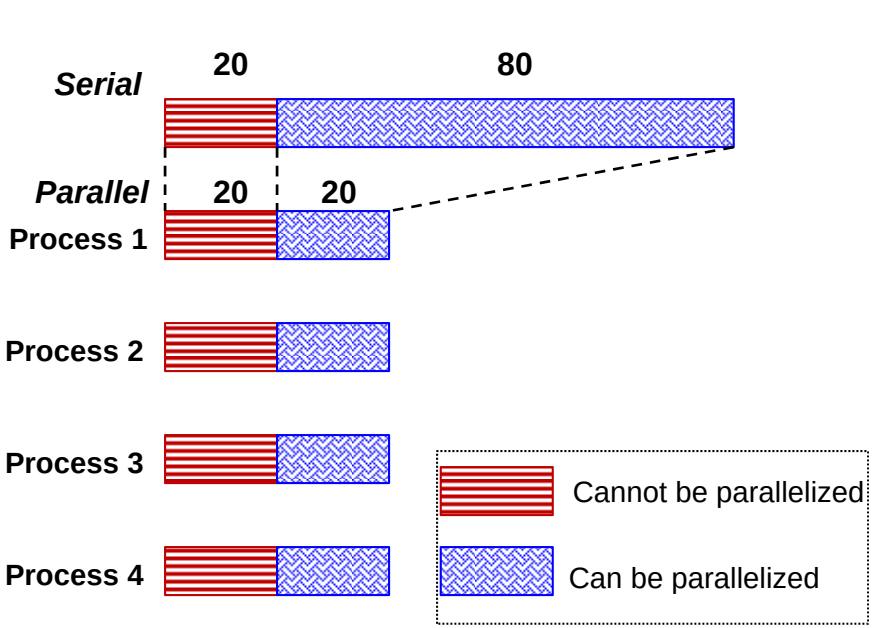
$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

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

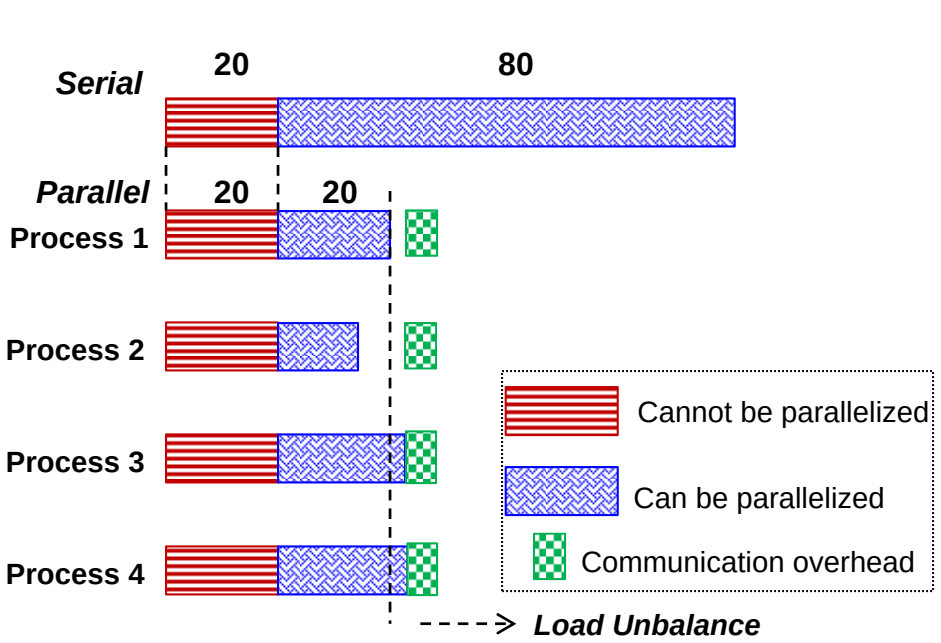
جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

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

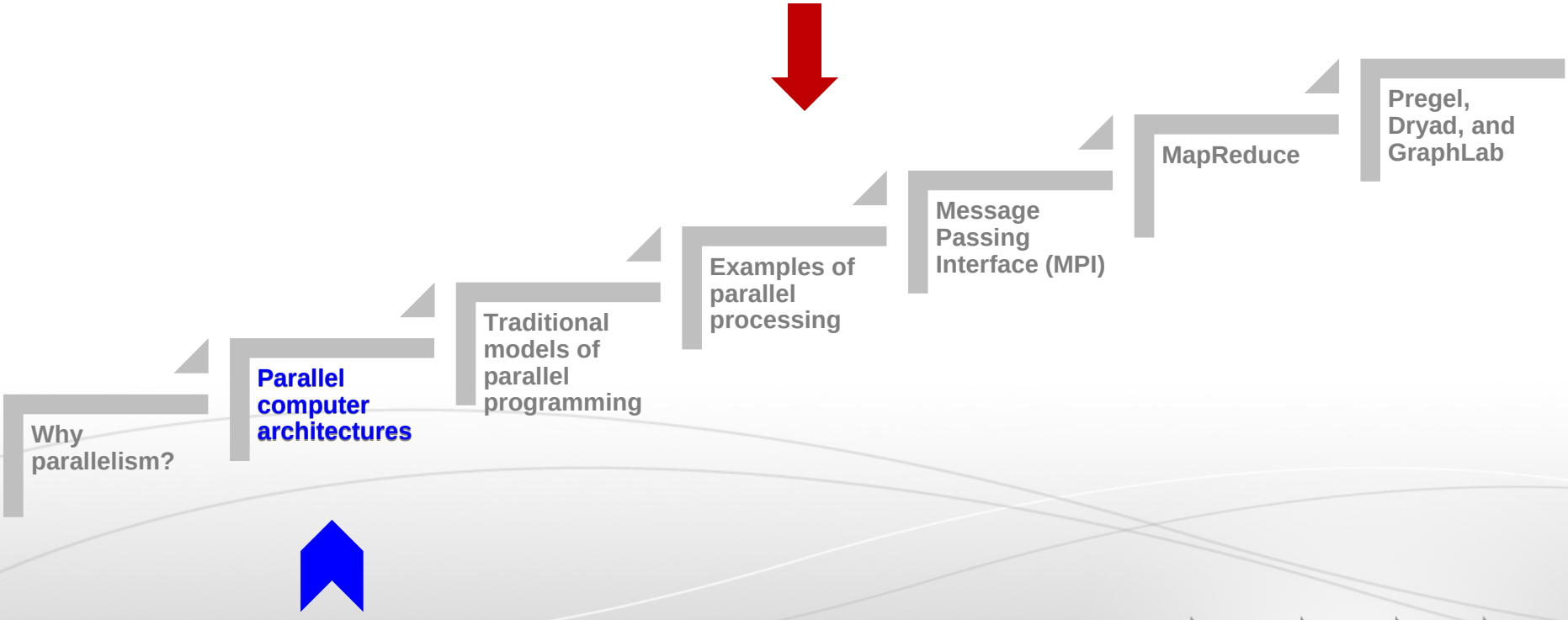
- In order to efficiently benefit from parallelization, we ought to follow these guidelines:

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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

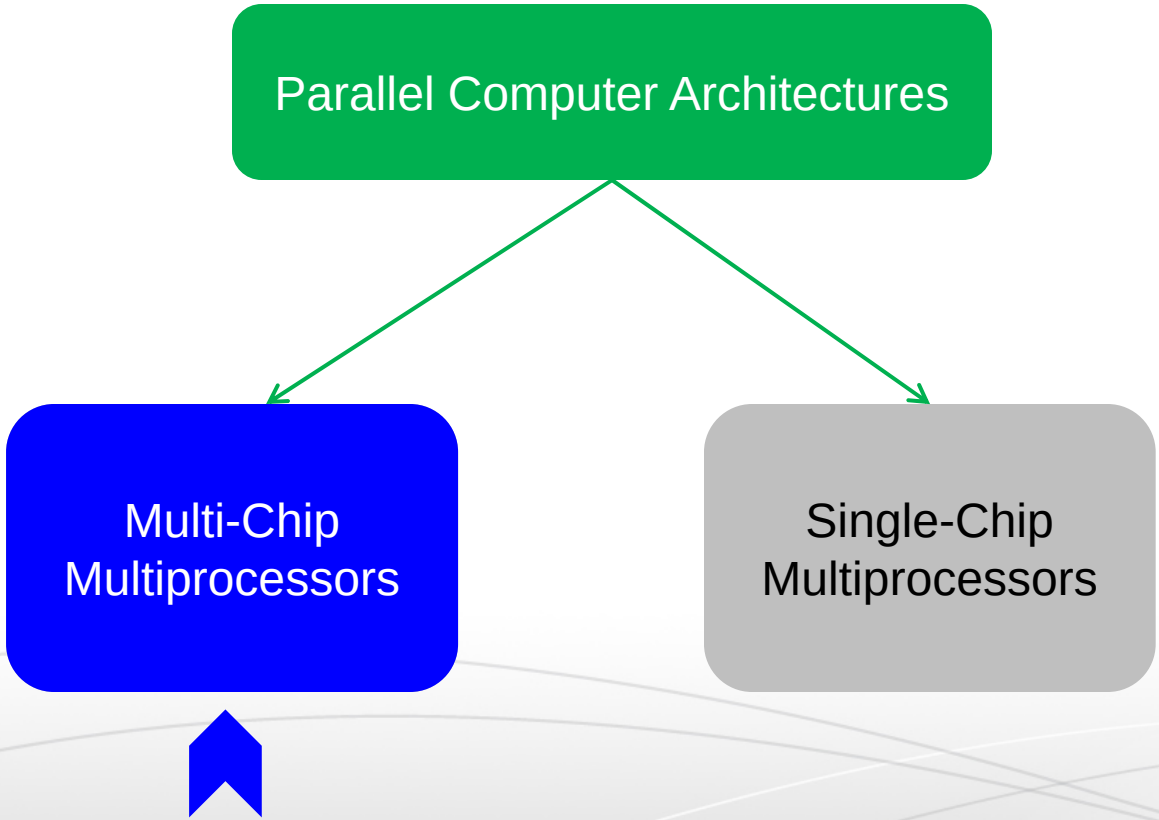
Objectives

Discussion on Programming Models



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Parallel Computer Architectures



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Multi-Chip Multiprocessors

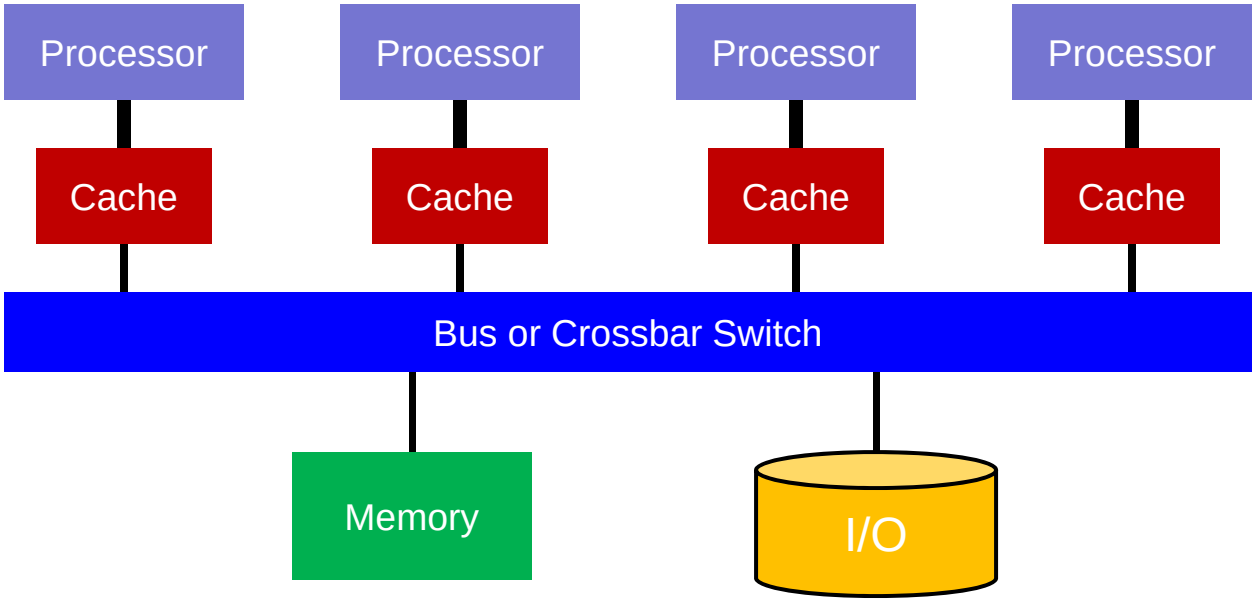
- We can categorize the architecture of multi-chip multiprocessor computers in terms of two aspects:
  - Whether the memory is physically centralized or distributed
  - Whether or not the address space is shared

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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Symmetric Multiprocessors

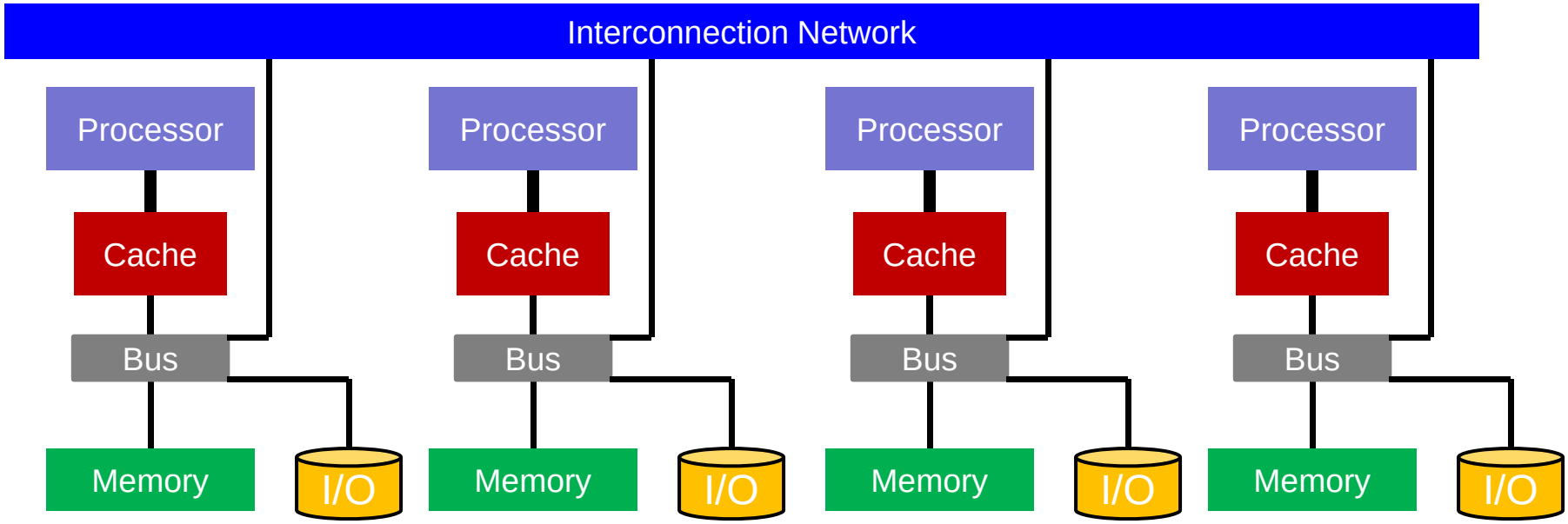
- A system with Symmetric Multiprocessors (SMP) architecture uses a shared memory that can be accessed equally from all processors



- Usually, a single OS controls the SMP system

Massively Parallel Processors

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



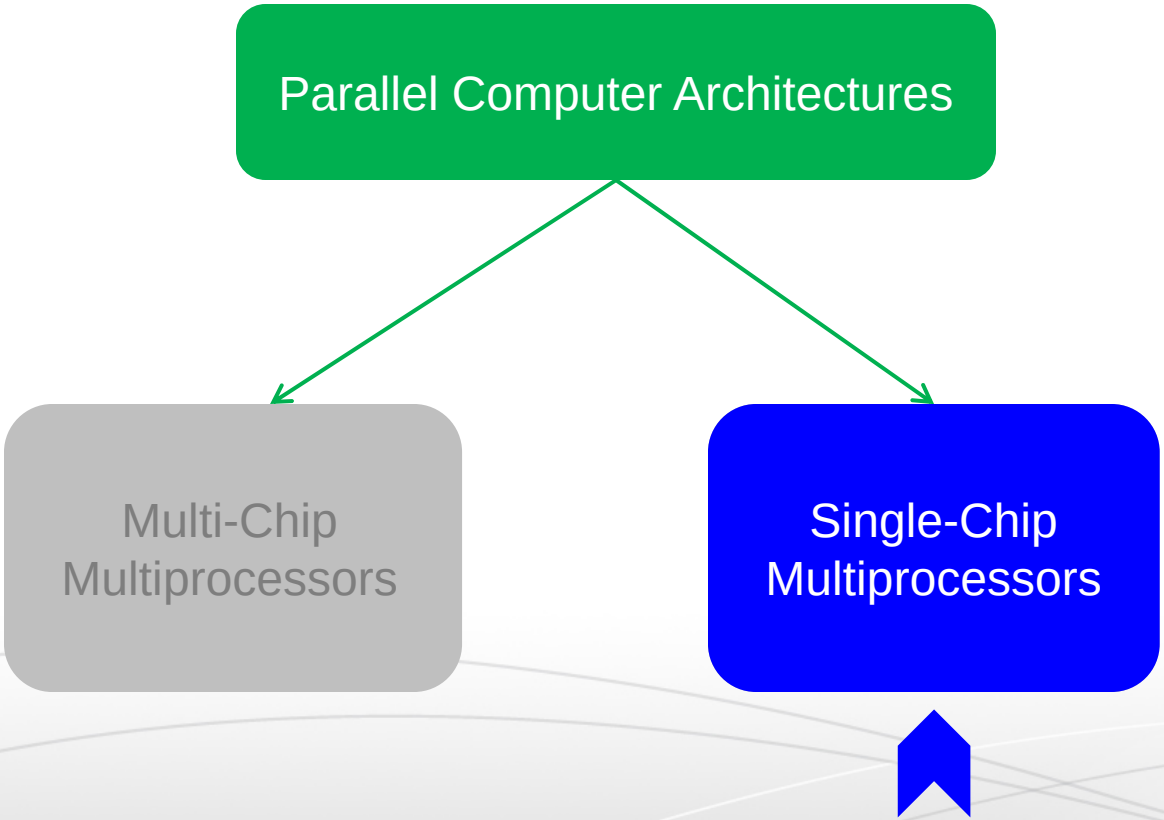
- Typically, an independent OS runs at each node

Distributed Shared Memory

- A Distributed Shared Memory (DSM) system is typically built on a similar hardware model as MPP
- DSM provides a shared address space to applications using a hardware/software directory-based coherence protocol
- The memory latency varies according to whether the memory is accessed directly (a local access) or through the interconnect (a remote access) (hence, NUMA)
- As in a SMP system, typically a single OS controls a DSM system

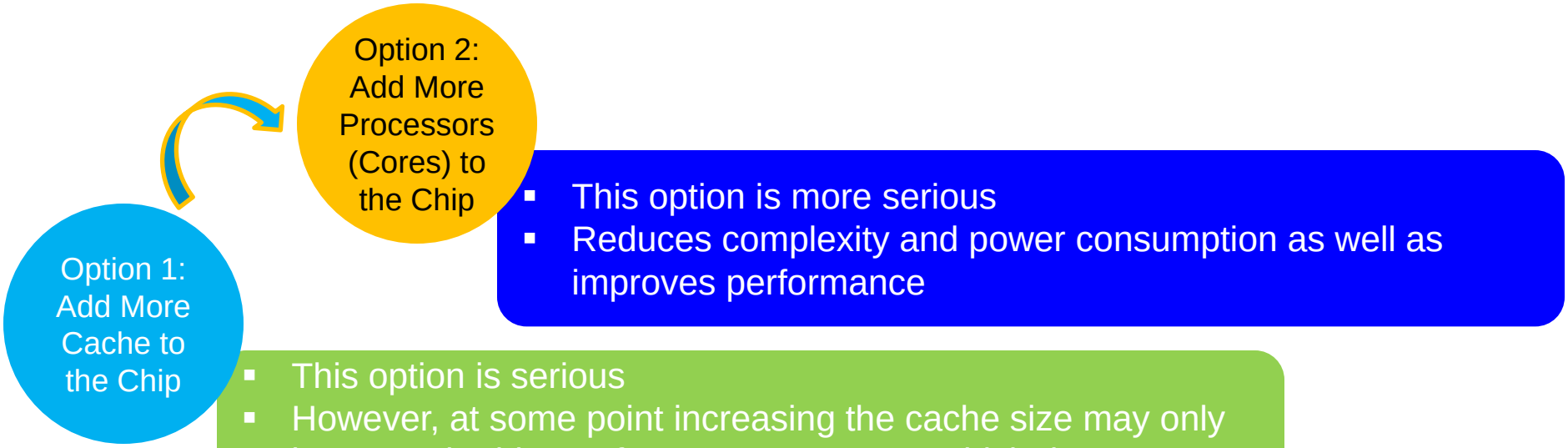
جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Parallel Computer Architectures



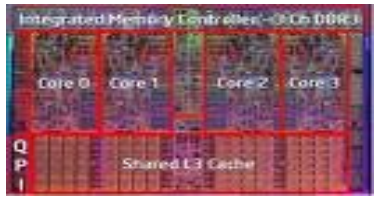
Moore's Law

- As chip manufacturing technology improves, transistors are getting smaller and smaller and it is possible to put more of them on a chip
- This empirical observation is often called Moore's Law (# of transistors doubles every 18 to 24 months)
- An obvious question is: "What do we do with all these transistors"?

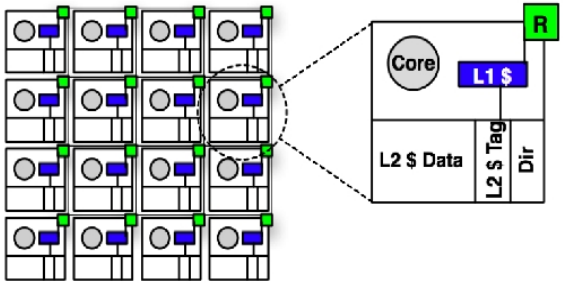


Chip Multiprocessors

- The outcome is a single-chip multiprocessor referred to as Chip Multiprocessor (CMP)



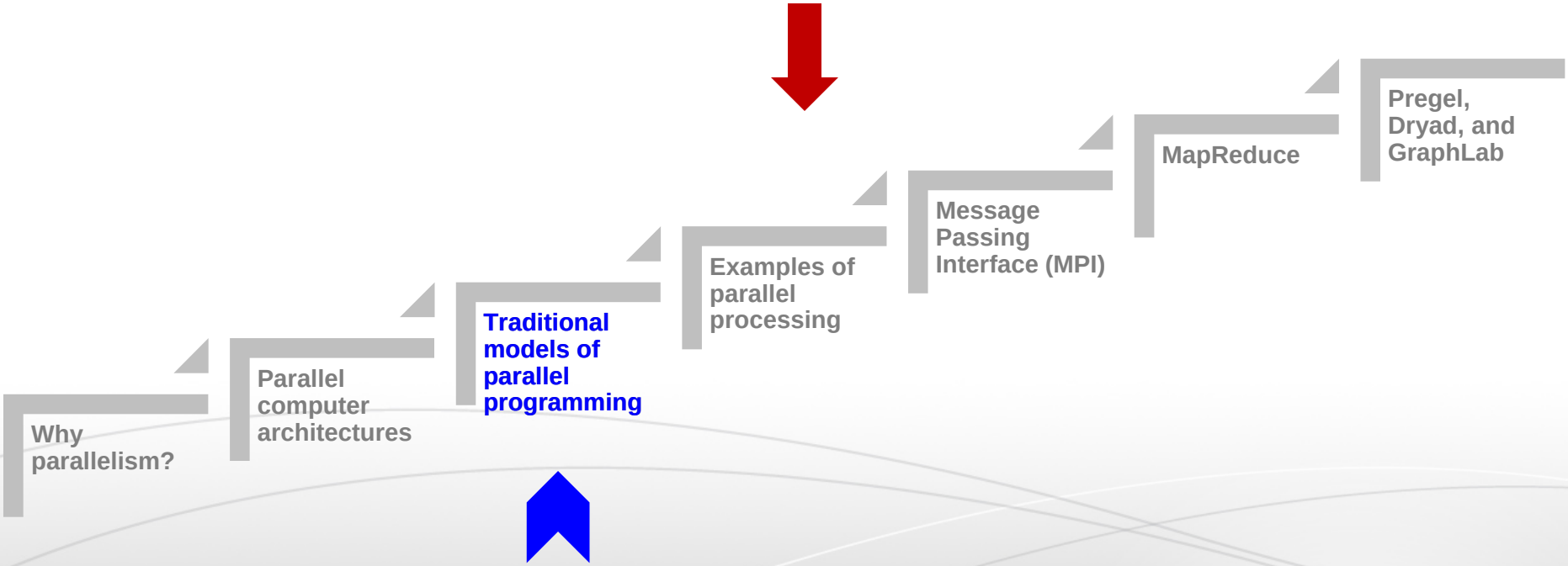
- CMP is currently considered the architecture of choice
- Cores in a CMP might be coupled either tightly or loosely
  - Cores may or may not share caches
  - Cores may implement a message passing or a shared memory inter-core communication method
- Common CMP interconnects (referred to as Network-on-Chips or NoCs) include bus, ring, 2D mesh, and crossbar



- CMPs could be homogeneous or heterogeneous:
  - Homogeneous CMPs include only identical cores
  - Heterogeneous CMPs have cores which are not identical

Objectives

Discussion on Programming Models



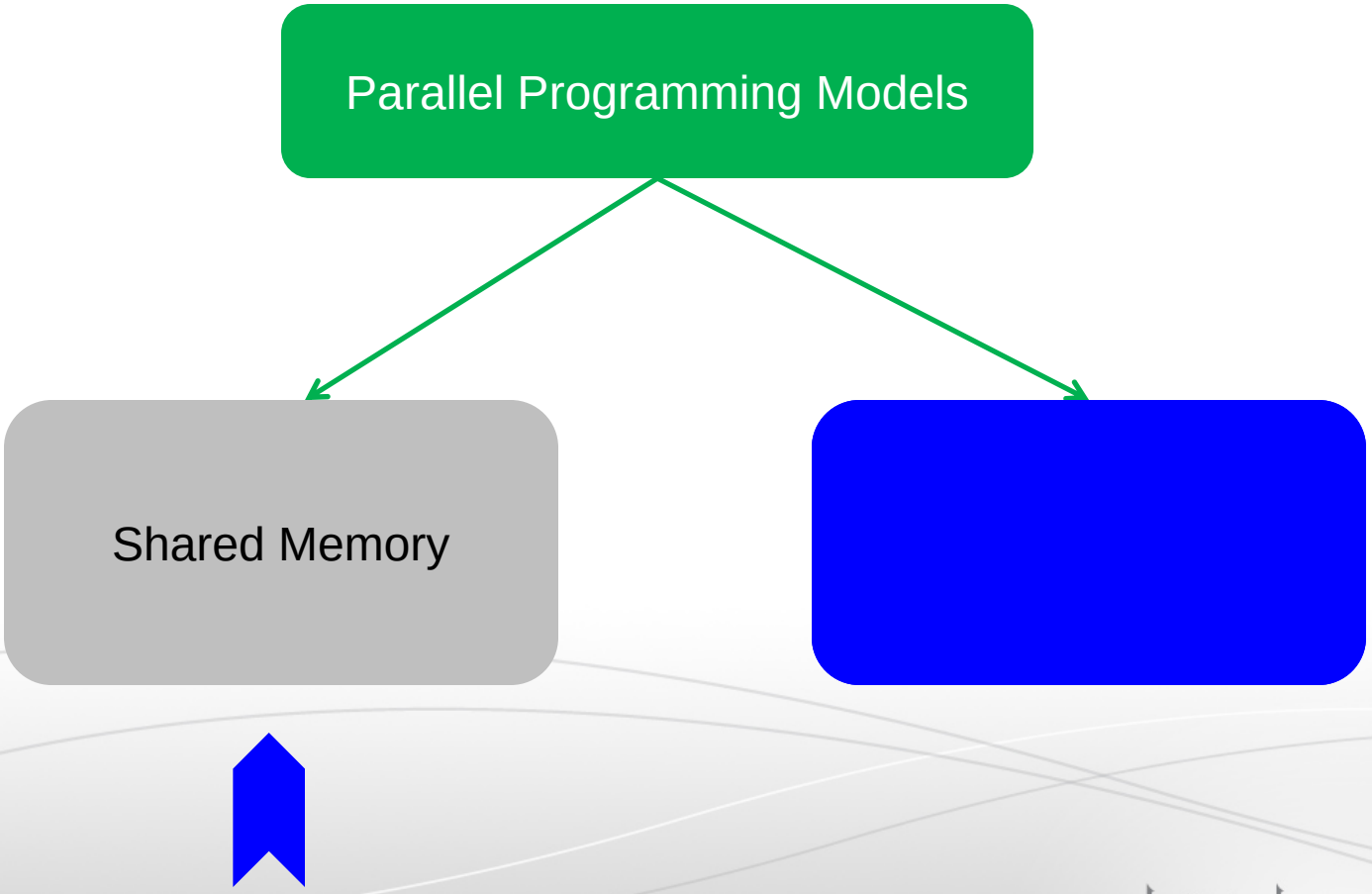
جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

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, CMP) and minimize the elapsed time of your program

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Traditional Parallel Programming Models

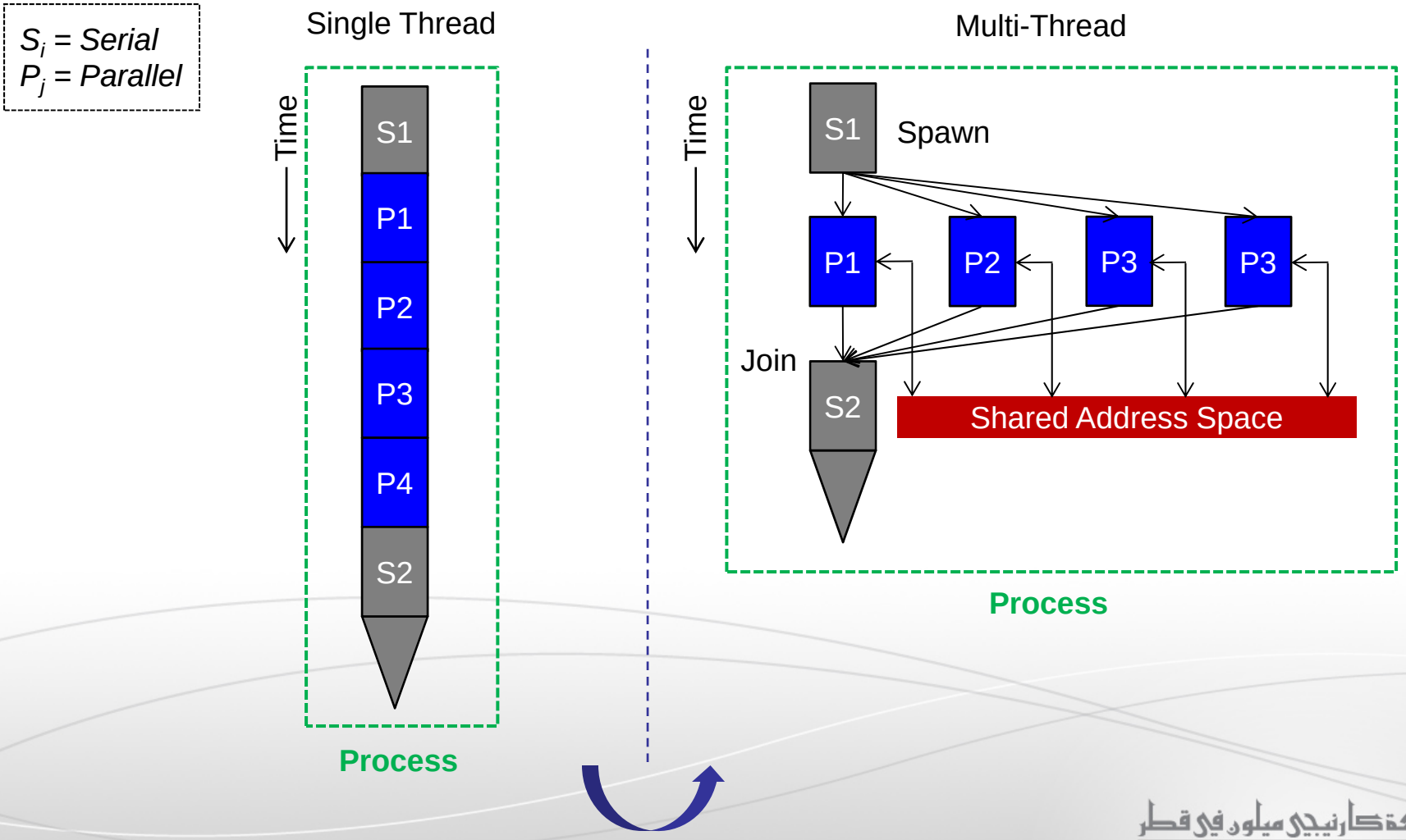


Shared Memory Model

- 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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Shared Memory Model



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

Why Locks?

- Unfortunately, threads in a shared memory model need to synchronize
- This is usually achieved through mutual exclusion
- Mutual exclusion requires that when there are multiple threads, only one thread is allowed to write to a shared memory location (or the critical section) at any time
- How to guarantee mutual exclusion in a critical section?
  - Typically, a lock can be implemented

```
//In a high level language
void lock (int *lockvar) {
  wh
  *l
}
void
  *lockvar = 0;
}
```

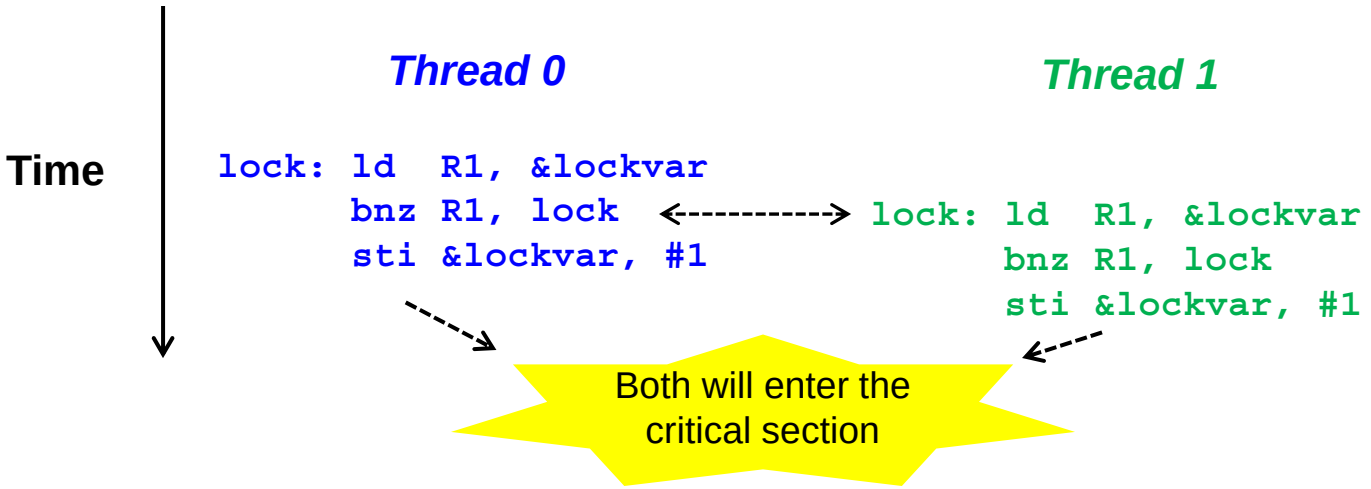
In machine language, it looks like this:

```
unlock: st   &lockvar, #0
        ret
```

Is this Enough/Correct?

The Synchronization Problem

- Let us check if this works:



- The execution of **ld**, **bnz**, and **sti** is not atomic (or indivisible)
  - Several threads may be executing them at the same time
- This allows several threads to enter the critical section simultaneously

The Peterson's Algorithm

- To solve this problem, let us consider a software solution referred to as the Peterson's Algorithm [Tanenbaum, 1992]

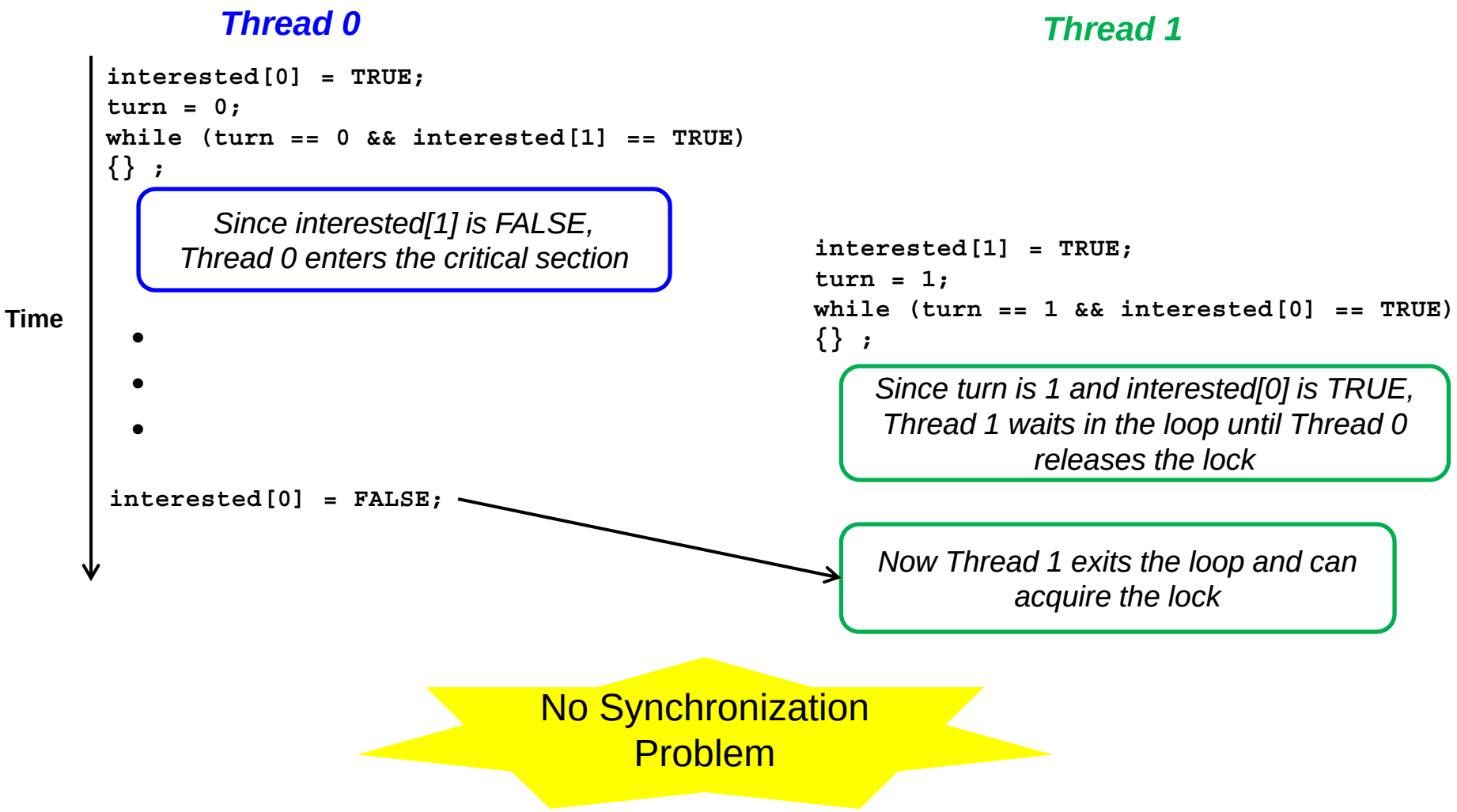
```
int turn;
int interested[n];  // initialized to 0

void lock (int process, int lvar) {     // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process, int lvar) {
  interested[process] = FALSE;
}
```

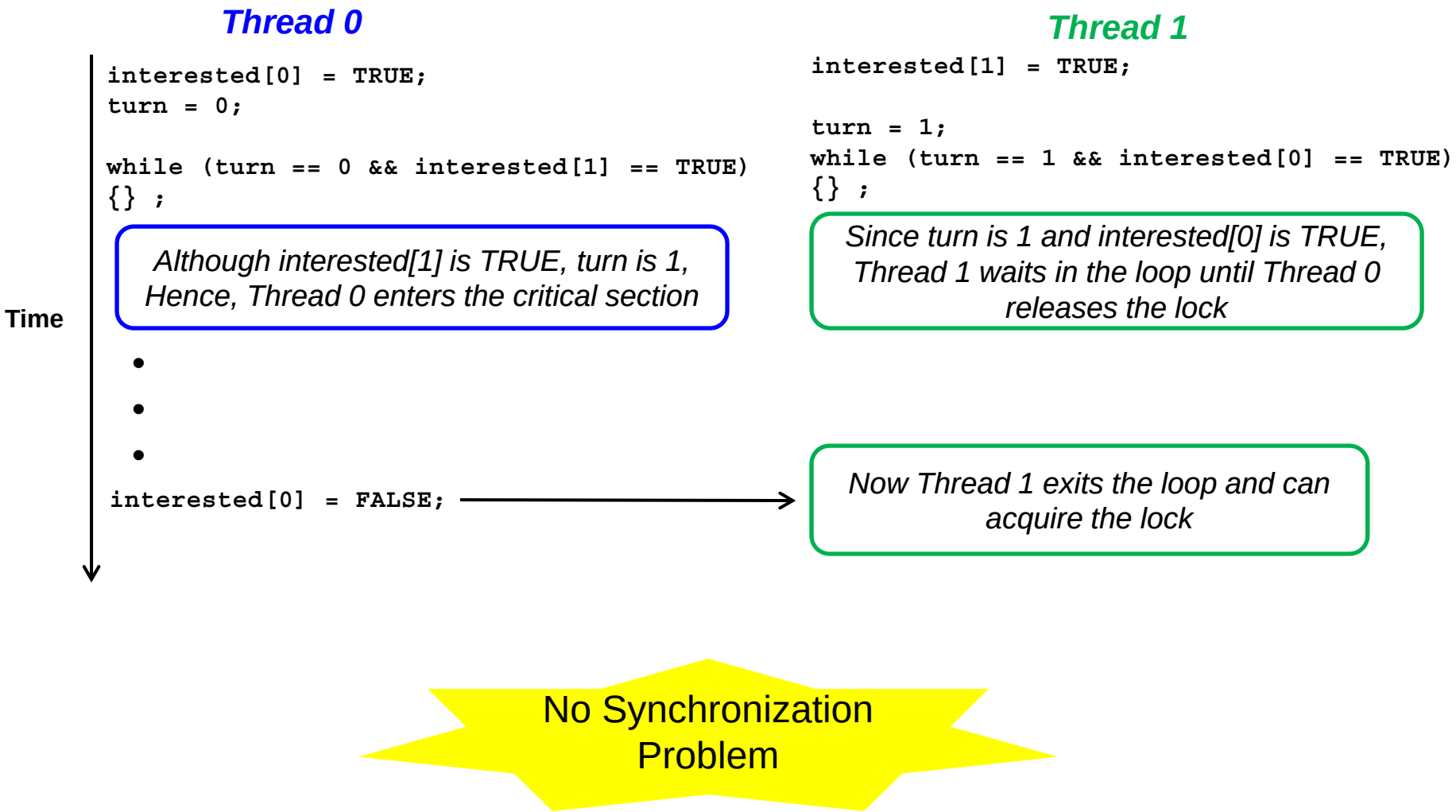
جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

No Race



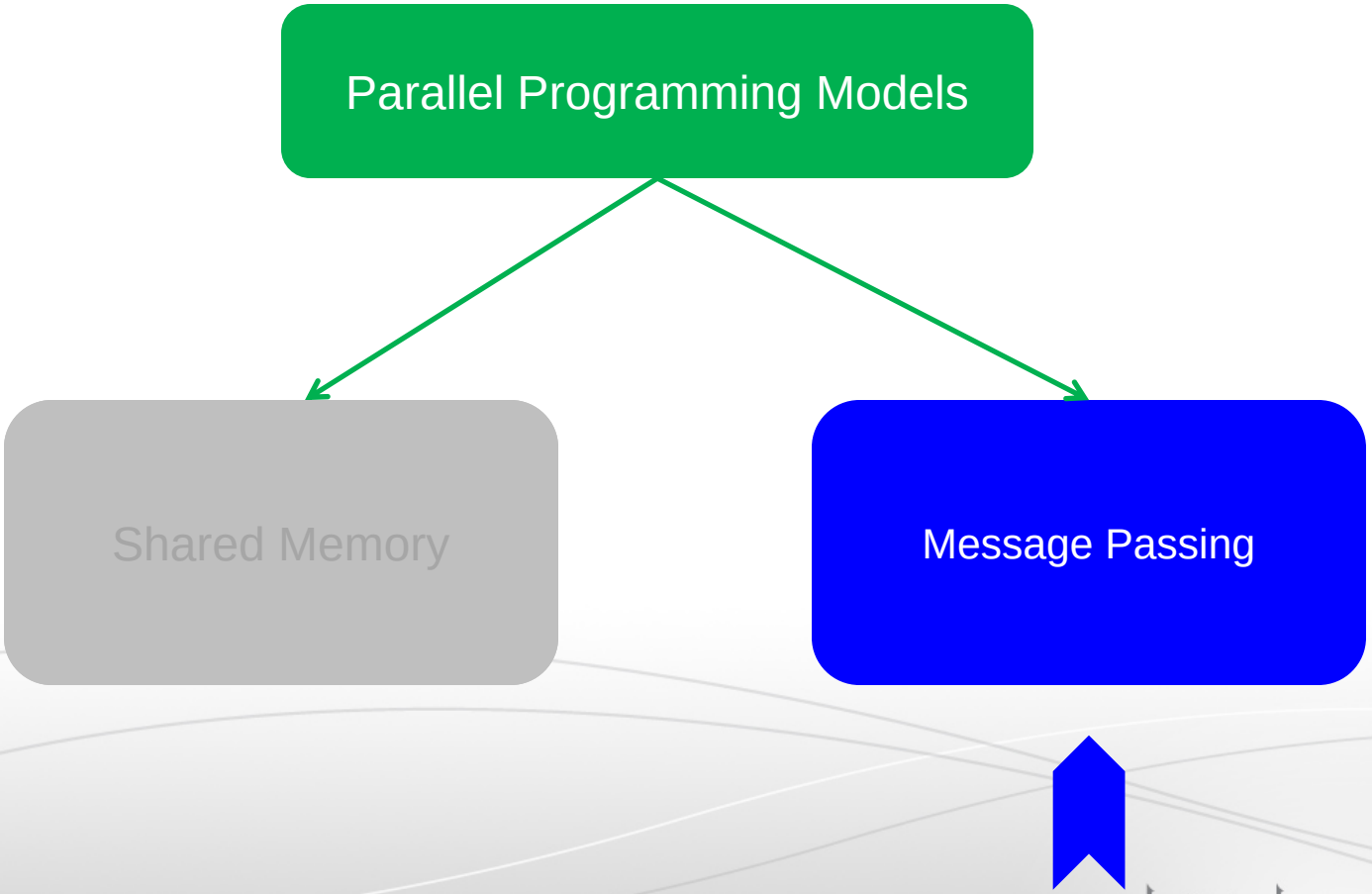
No Synchronization Problem

With Race



No Synchronization Problem

Traditional Parallel Programming Models

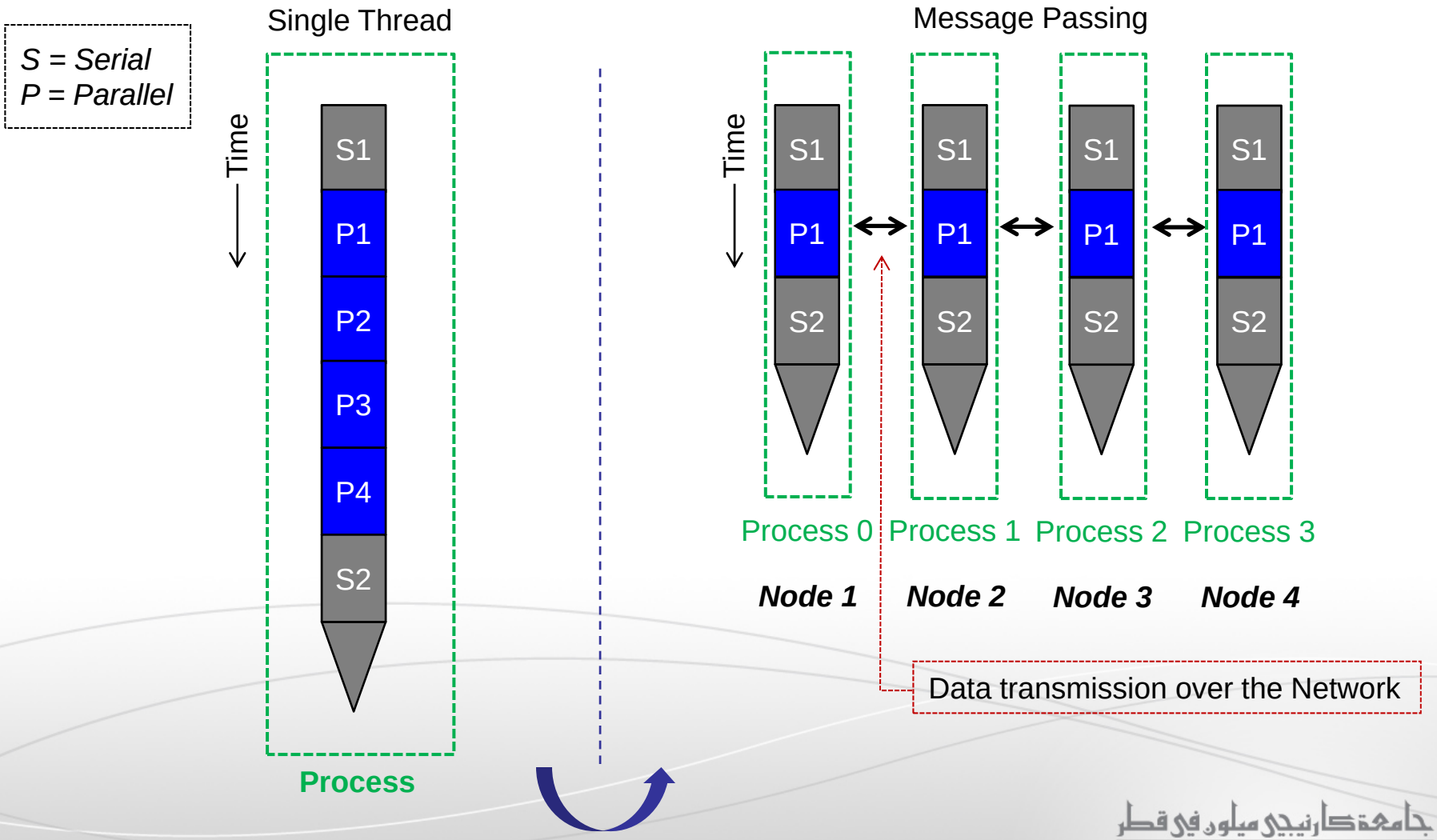


Message Passing Model

- 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
- Message Passing Interface (MPI) programs are the best fit with the message passing programming model

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Message Passing Model



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
  re

for
  a[

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);
```

No Mutual Exclusion is Required!

Parallel

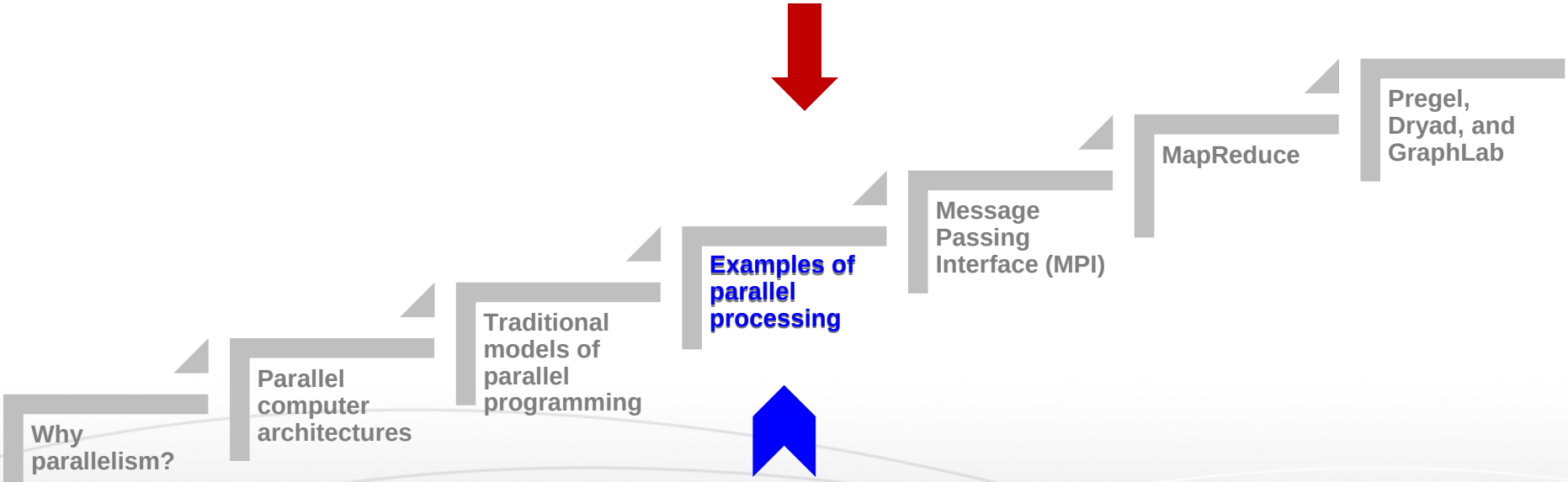
Shared Memory Vs. Message Passing

- 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



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

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:

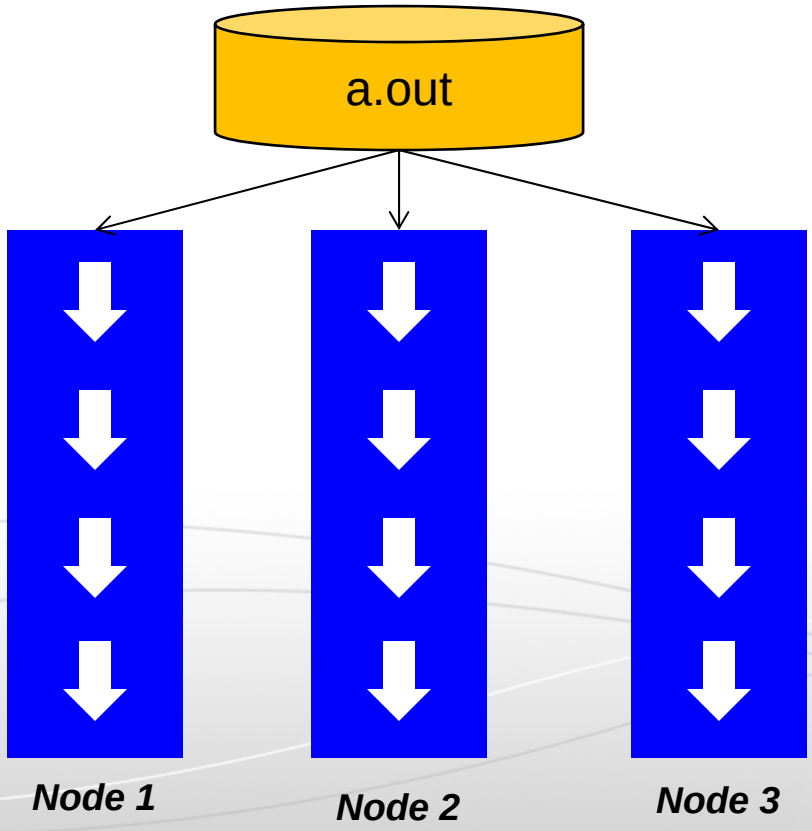
  1. Single Program Multiple Data (**SPMD**) model

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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

SPMD

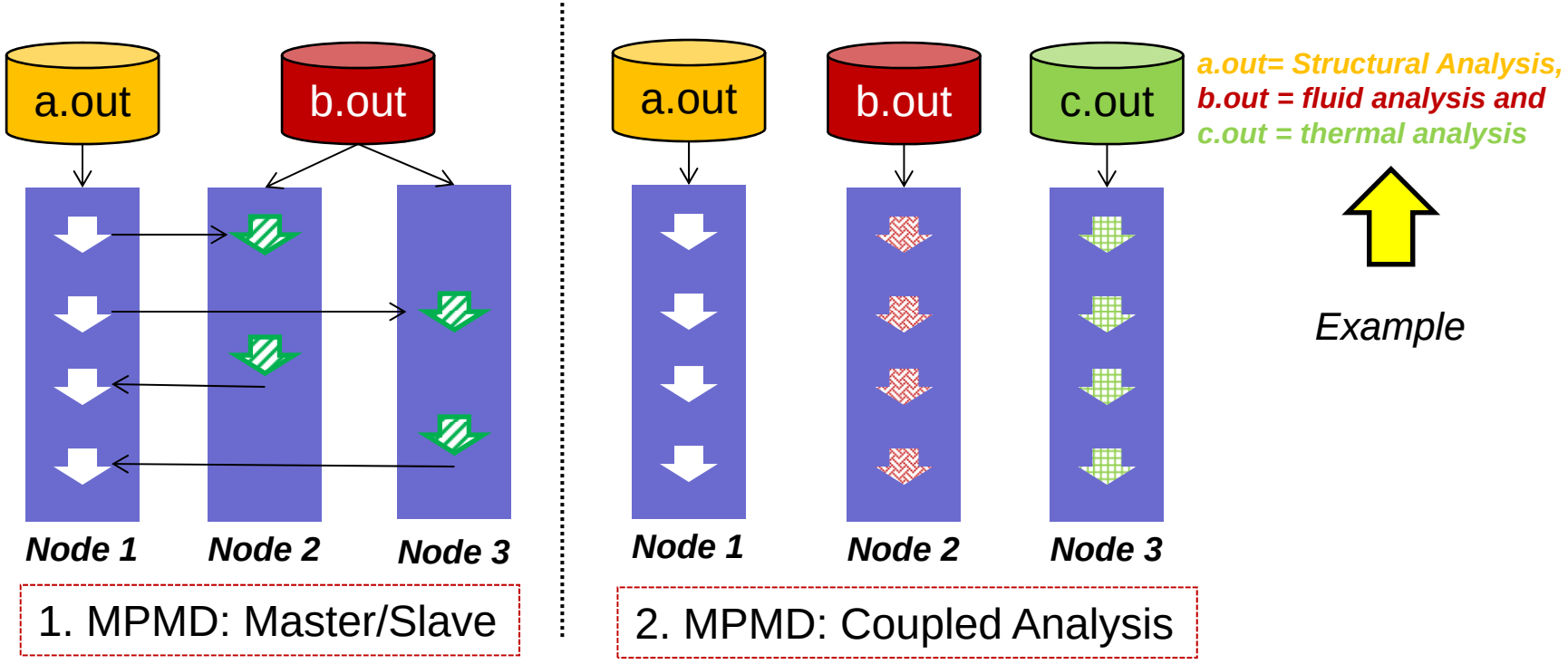
- In the SPMD model, there is only one program and each process uses the same executable working on different sets of data



MPMD

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

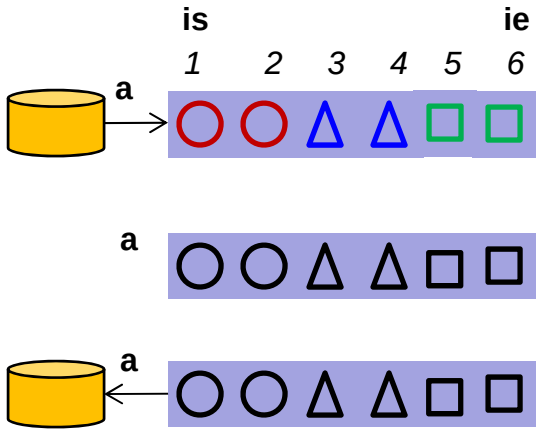
- MPMD has two styles, the *master/worker* and the *coupled analysis*



An Example

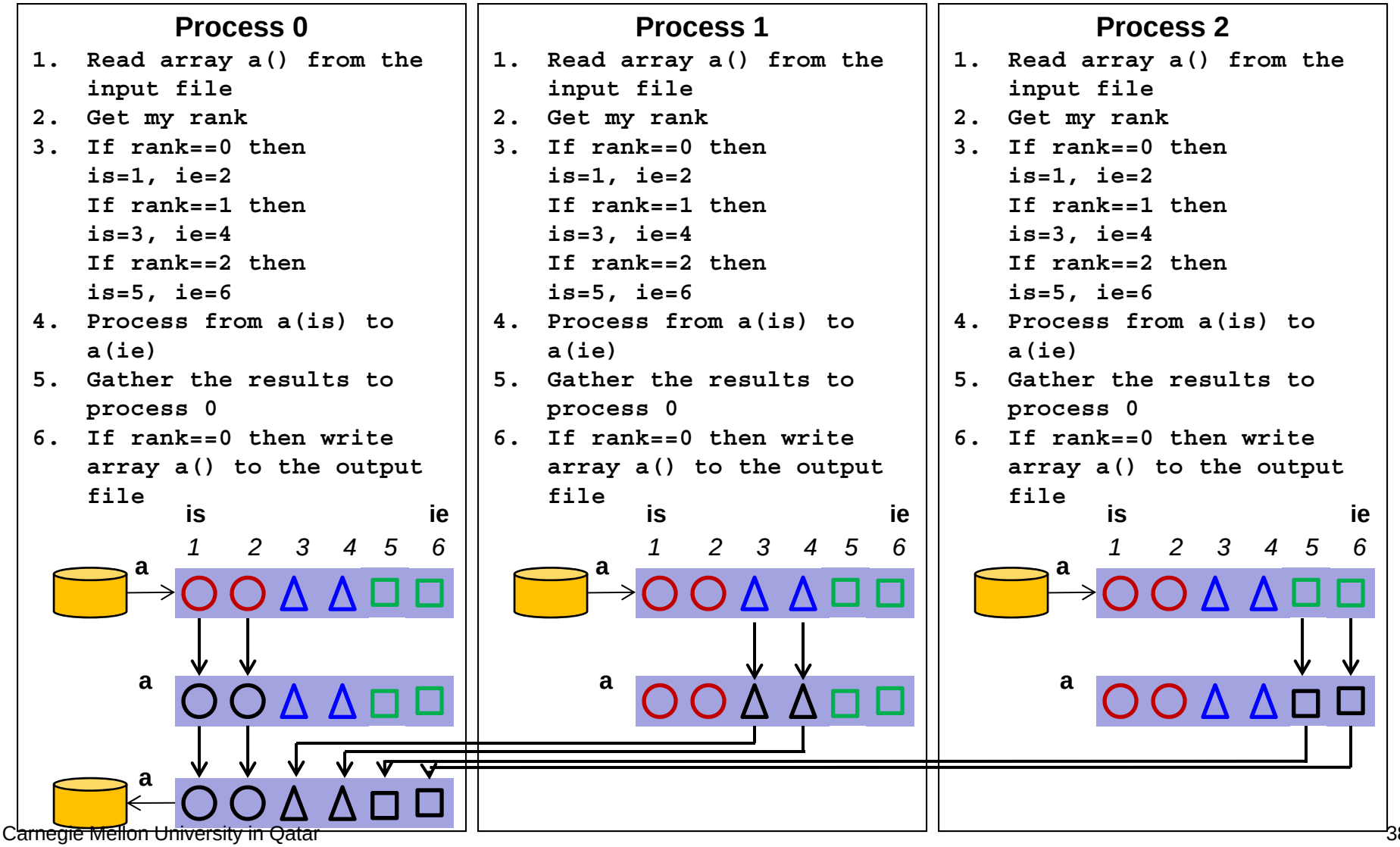
A Sequential Program

```
1. Read array a() from the input file
2. Set is=1 and ie=6  //is = index start and ie = index end
3. Process from a(is) to a(ie)
4. Write array a() to the output file
```



- Colored shapes indicate the initial values of the elements
- Black shapes indicate the values after they are processed

An Example



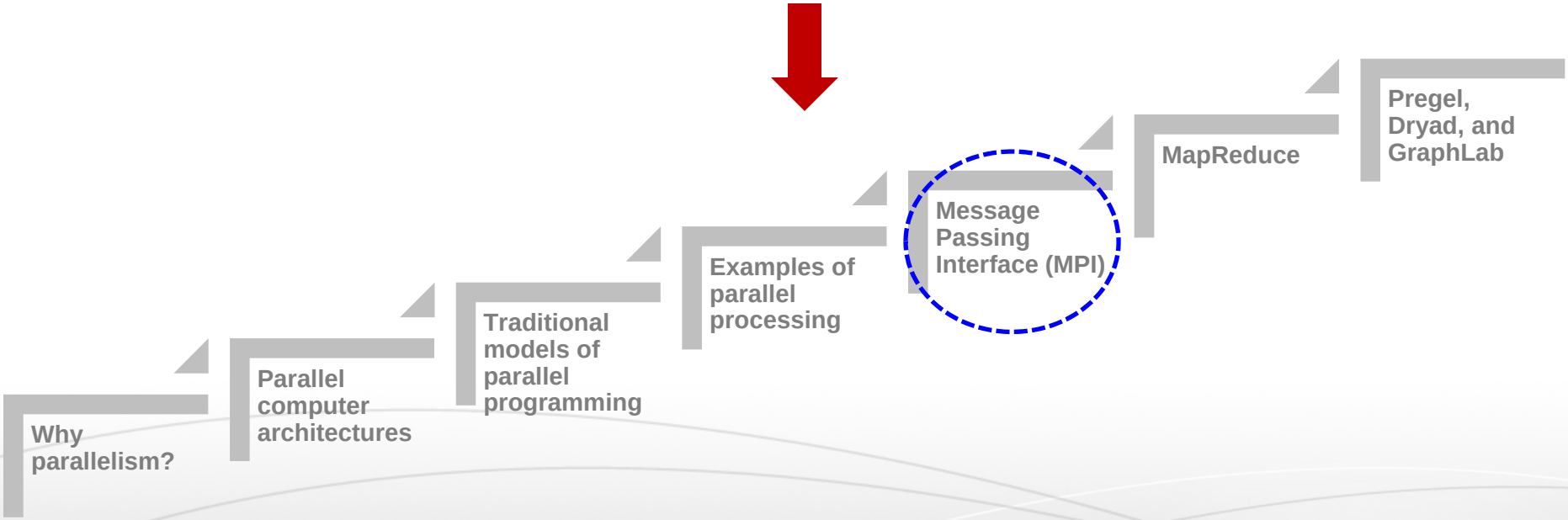
Concluding Remarks

- 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

جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar

Next Class

Discussion on Programming Models



جامعة كارنيجي ميلون في قطر
Carnegie Mellon Qatar