



香港中文大學  
The Chinese University of Hong Kong

CENG3420

## Lecture 11: Multi-Threading & Multi-Core

**Bei Yu**

(Latest update: April 16, 2020)

Spring 2020

# Overview

Introduction

Amdahl's Law

Thread-Level Parallelism (TLP)

Multi-Cores



# Overview

Introduction

Amdahl's Law

Thread-Level Parallelism (TLP)

Multi-Cores



# Limits to ILP

Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to

- ▶ issue 3 or 4 data memory accesses per cycle,
- ▶ resolve 2 or 3 branches per cycle,
- ▶ rename and access more than 20 registers per cycle, and
- ▶ fetch 12 to 24 instructions per cycle.

The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate

- ▶ E.g, widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power



# Overview

Introduction

**Amdahl's Law**

Thread-Level Parallelism (TLP)

Multi-Cores

# Encountering Amdahl's Law

Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

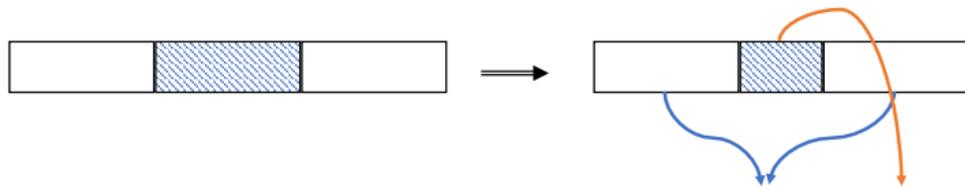


# Encountering Amdahl's Law

Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

Suppose that enhancement E accelerates a fraction  $F$  ( $F < 1$ ) of the task by a factor  $S$  ( $S > 1$ ) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} * ((1-F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$



## Example 1: Amdahl's Law

Consider an enhancement which runs 20 times faster but which is only usable 25% of the time.

Speedup w/ E =

What is its usable only 15% of the time?

Speedup w/ E =



# Scalar v.s. Vector

- ▶ A scalar processor processes only **one datum** at a time.
- ▶ A vector processor implements an instruction set containing instructions that operate on **one-dimensional** arrays of data called **vectors**.



- ▶ To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less
- ▶ Amdahl's Law tells us that to achieve linear speedup with 100 processors, none of the original computation can be scalar!



## Example 2: Amdahl's Law

Consider 10 scalar variable summings and two 10 by 10 matrices (matrix sum) on 10 processors

Speedup w/ E =

What if there are 100 processors ?

Speedup w/ E =

What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?

Speedup w/ E =

What if there are 100 processors ?

Speedup w/ E =



# Overview

Introduction

Amdahl's Law

Thread-Level Parallelism (TLP)

Multi-Cores

# Multi-Threading

- ▶ Difficult to continue to extract instruction-level parallelism (ILP) from a single sequential thread of control
  - ▶ Many workloads can make use of thread-level parallelism (TLP)
- ▶ TLP from [multiprogramming](#) (run independent sequential jobs)
  - ▶ TLP from [multithreaded applications](#) (run one job faster using parallel threads)
- ▶ Multithreading uses TLP to improve utilization of a single processor



# Examples of Threads

## A web browser

- ▶ One thread displays images
- ▶ One thread retrieves data from network

## A word processor

- ▶ One thread displays graphics
- ▶ One thread reads keystrokes
- ▶ One thread performs spell checking in the background

## A web server

- ▶ One thread accepts requests
- ▶ When a request comes in, separate thread is created to service
- ▶ Many threads to support thousands of client requests



# Multi-Threading on A Chip

Find a way to “hide” true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are independent of those stalling instructions

## Hardware Multithreading

Increase the utilization of resources on a chip by allowing multiple processes (threads) to share the functional units of a single processor

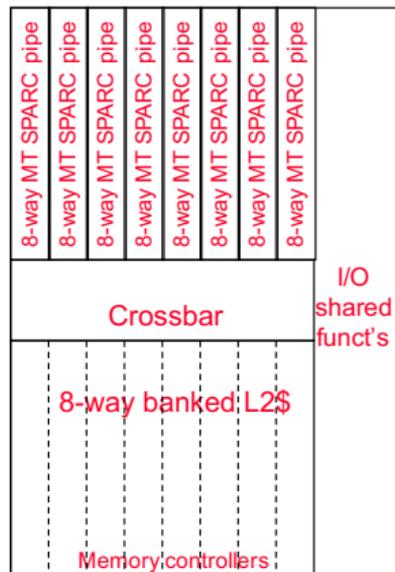
- ▶ Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
- ▶ The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
- ▶ The memory can be shared through virtual memory mechanisms
- ▶ Hardware must support efficient thread context switching



# Multithreaded Example: Sun's Niagara (UltraSparc T2)

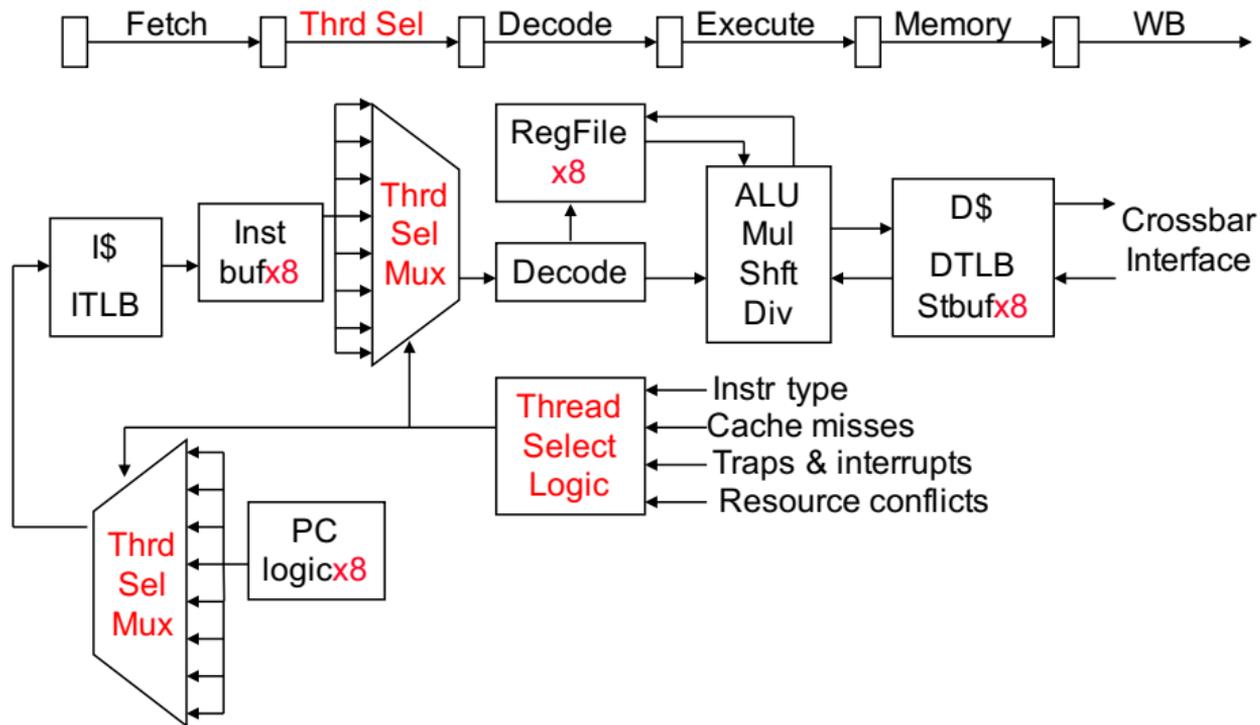
Eight fine grain multithreaded single-issue, in-order cores (no speculation, no dynamic branch prediction)

|                |             |
|----------------|-------------|
|                | Niagara 2   |
| Data width     | 64-b        |
| Clock rate     | 1.4 GHz     |
| Cache (I/D/L2) | 16K/8K/4M   |
| Issue rate     | 1 issue     |
| Pipe stages    | 6 stages    |
| BHT entries    | None        |
| TLB entries    | 64I/64D     |
| Memory BW      | 60+ GB/s    |
| Transistors    | ??? million |
| Power (max)    | <95 W       |



# Niagara Integer Pipeline

Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient



# Types of Multithreading

## Coarse-grain

Switches threads only on costly stalls (e.g., L2 cache misses)

- ▶ 😊 Thread switching doesn't have to be essentially free and much less likely to slow down the execution of an individual thread
- ▶ 😞 Limited, due to pipeline start-up costs, in its ability to overcome throughput loss
- ▶ 😞 Pipeline must be flushed and refilled on thread switches

## Fine-grain

Switch threads on every instruction issue

- ▶ Round-robin thread interleaving (skipping stalled threads)
- ▶ Processor must be able to switch threads on every clock cycle
- ▶ 😊 Can hide throughput losses that come from both short and long stalls
- ▶ 😞 Slows down the execution of an individual thread since a thread that is ready to execute without stalls is delayed by instructions from other threads



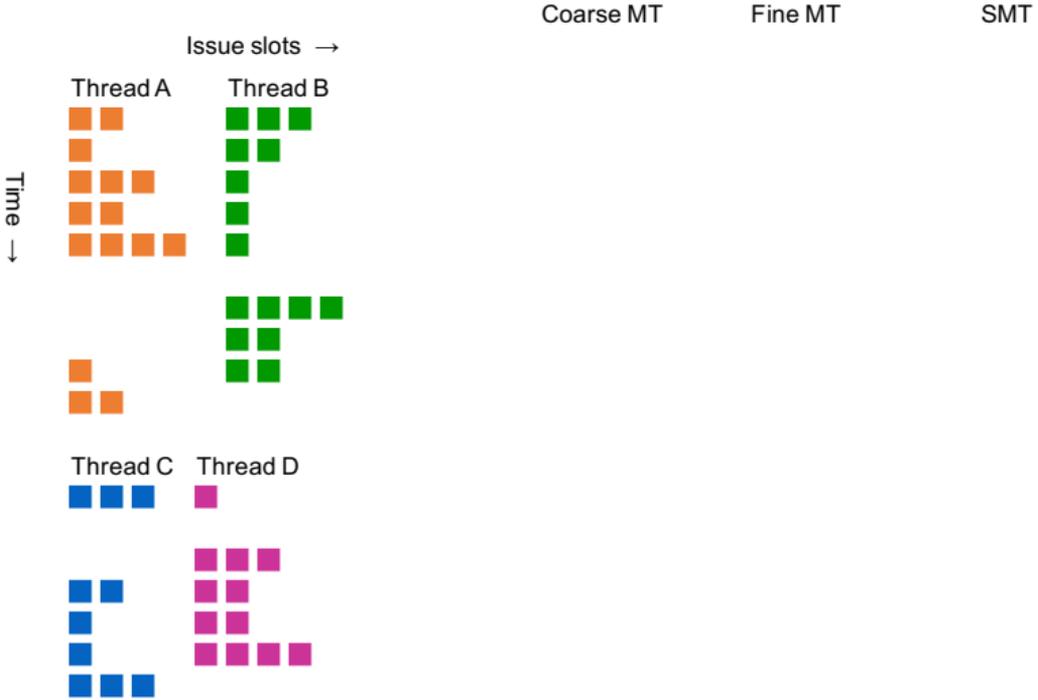
# Simultaneous Multithreading (SMT)

A variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor (superscalar) to exploit both program ILP and TLP

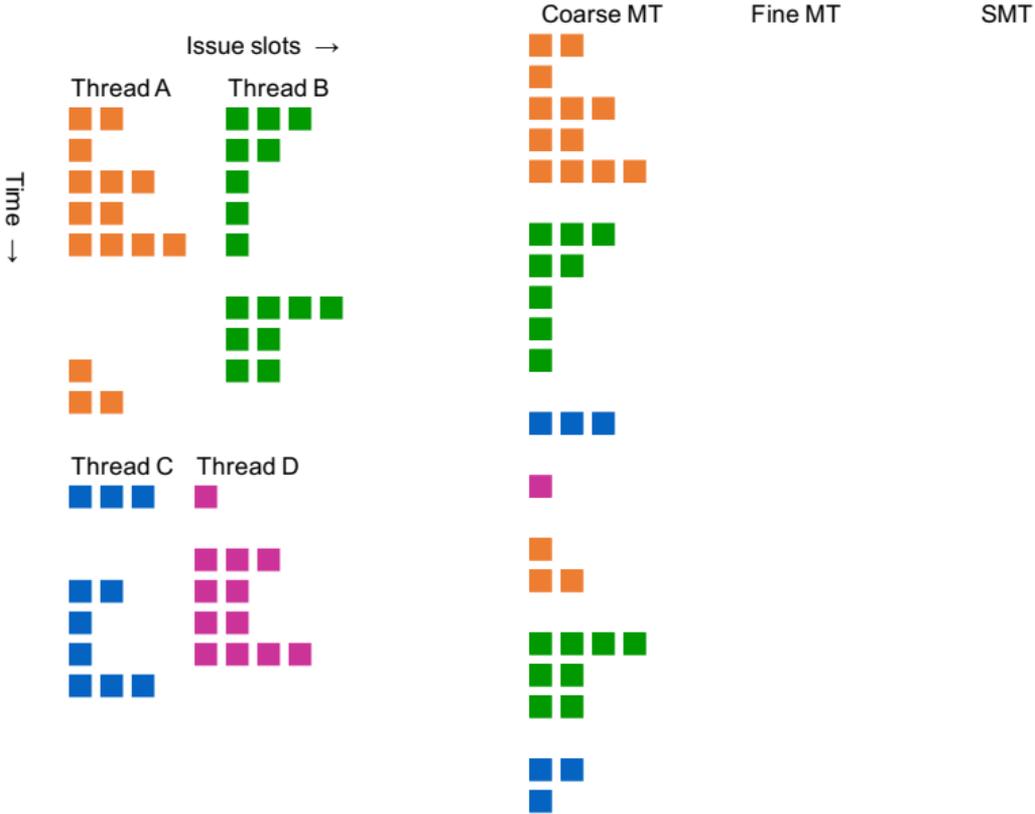
- ▶ Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)
- ▶ With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them
- ▶ Need separate rename tables (RUUs) for each thread or need to be able to indicate which thread the entry belongs to
- ▶ Need the capability to commit from multiple threads in one cycle
- ▶ Intel's Pentium 4 SMT is called **hyperthreading**: supports just two threads (doubles the architecture state)



# Threading on a 4-way SS Processor Example

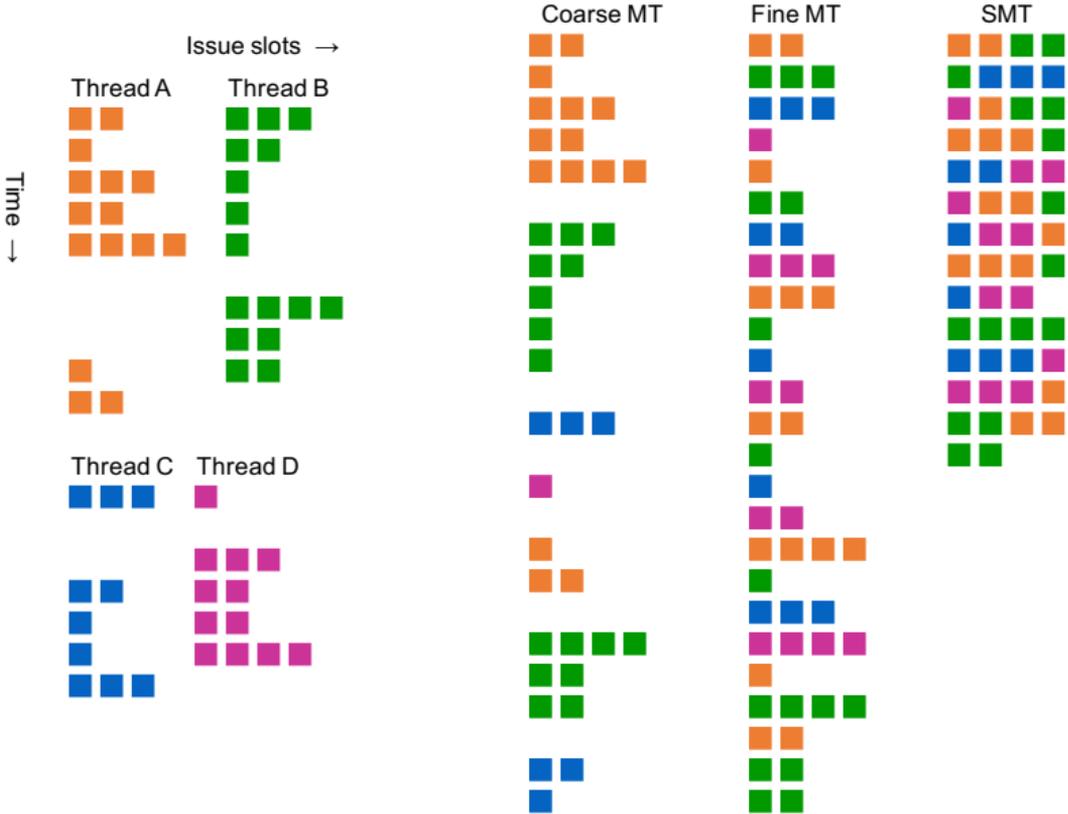


# Threading on a 4-way SS Processor Example





# Threading on a 4-way SS Processor Example



# Overview

Introduction

Amdahl's Law

Thread-Level Parallelism (TLP)

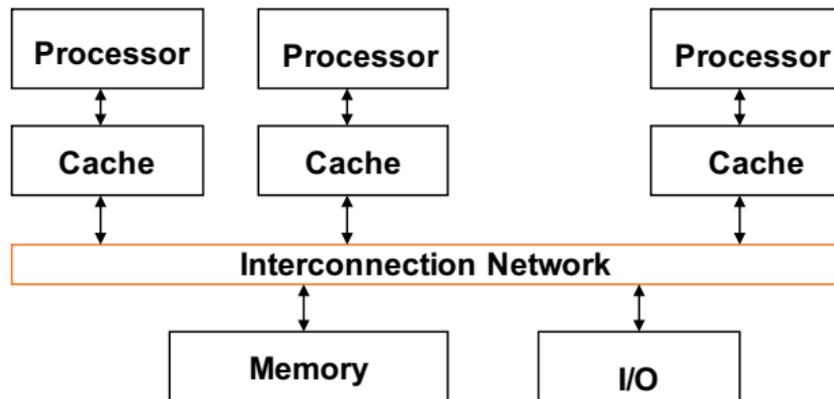
**Multi-Cores**



# The Big Picture: Where are We Now?

## Multiprocessor

A computer system with at least two processors



- ▶ Can deliver high throughput for independent jobs via [job-level parallelism](#) or [process-level parallelism](#)
- ▶ And improve the run time of a single program that has been specially crafted to run on a multiprocessor – a [parallel processing program](#)



# Multicores Now Universal

- ▶ Power challenge has forced a change in microprocessor design
- ▶ Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- ▶ Today's microprocessors typically contain more than one core – **Chip Multicore microProcessors (CMPs)** in a single IC

| Product        | AMD Barcelona | Intel Nehalem | IBM Power 6 | Sun Niagara 2 |
|----------------|---------------|---------------|-------------|---------------|
| Cores per chip | 4             | 4             | 2           | 8             |
| Clock rate     | 2.5 GHz       | ~2.5 GHz?     | 4.7 GHz     | 1.4 GHz       |
| Power          | 120 W         | ~100 W?       | ~100 W?     | 94 W          |



# Other Multiprocessor Basics

- ▶ Some of the problems that need higher performance can be handled simply by using a **cluster**
- ▶ A set of independent servers (or PCs) connected over a local area network (LAN) functioning as a single large multiprocessor
- ▶ E.g.: Search engines, Web servers, email servers, databases ...

## Key Challenge

Craft parallel (concurrent) programs that have high performance on multiprocessors as the number of processors increase

E.g.: Scale Scheduling, load balancing, time for synchronization, overhead for communication



# Scaling

To get good speedup on a multiprocessor while keeping the problem size fixed is **harder** than getting good speedup by increasing the size of the problem.

- ▶ **Strong scaling** –when speedup can be achieved on a multiprocessor without increasing the size of the problem
- ▶ **Weak scaling** – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors

Load balancing is another important factor. Just a single processor with twice the load of the others cuts the speedup almost in half



# Multiprocessor/Clusters Key Questions

Q1: How do they share data?

Q2: How do they coordinate?

Q3: How scalable is the architecture? How many processors can be supported?



# Shared Memory Multiprocessor (SMP)

## Q1: How do they share data?

Single address space shared by all processors

## Q2: How do they coordinate?

Processors coordinate/communicate through **shared variables** in memory (via loads and stores)

- ▶ Shared data coordinated via **synchronization primitives** (locks) that allow access by only one processor at a time



# Shared Memory Multiprocessor (SMP)

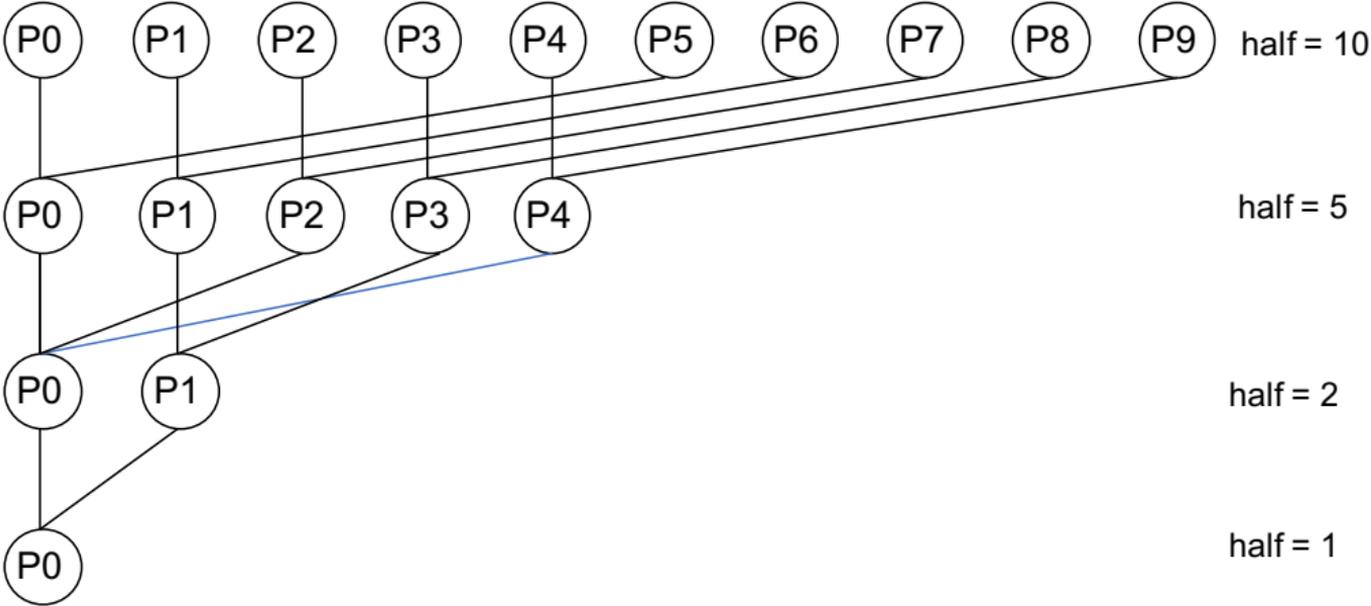
## 2 Multiprocessor Styles:

- ▶ Uniform memory access (UMA)
  - ▶ Nonuniform memory access (NUMA)
- 
- ▶ Programming NUMAs are harder
  - ▶ But NUMAs can scale to larger sizes and have lower latency to local memory



# An Example with 10 Processors

sum[P0]sum[P1]sum[P2] sum[P3]sum[P4]sum[P5]sum[P6] sum[P7]sum[P8] sum[P9]



# Process Synchronization

- ▶ Need to be able to coordinate processes working on a common task
- ▶ Lock variables ([semaphores](#)) are used to coordinate or synchronize processes

Need an architecture-supported [arbitration](#) mechanism

- ▶ decide which processor gets access to the lock variable
- ▶ [Single bus](#) provides arbitration mechanism, since the bus is the only path to memory
- ▶ The processor gets the bus wins

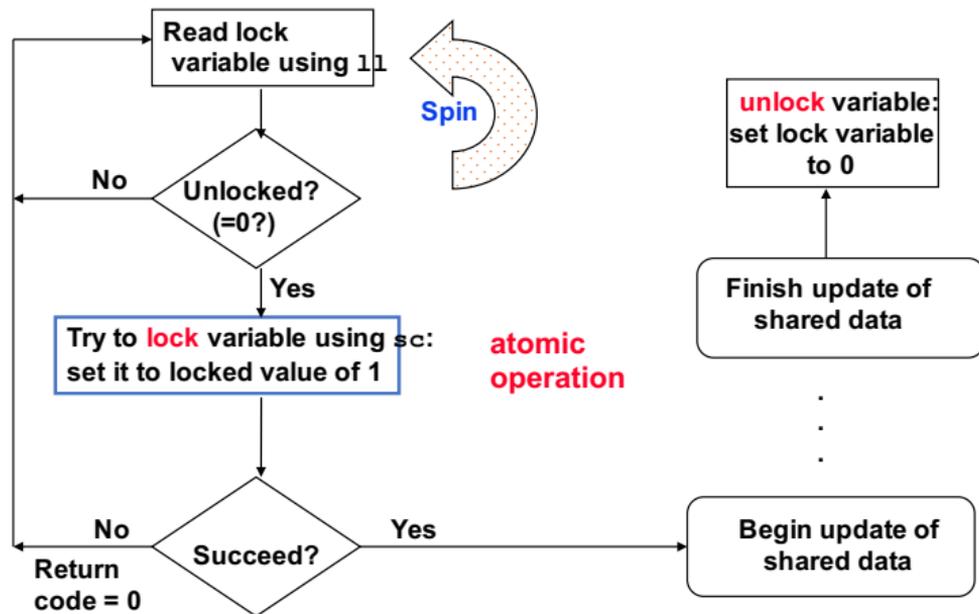
Need an architecture-supported [operation](#) that

- ▶ locks the variable
- ▶ Locking can be done via an [atomic swap operation](#)



# Spin Lock Synchronization

The single **winning** processor will succeed in writing a **1** to the lock variable; all others processors will get a return code of **0**



# Summing 100,000 Numbers on 100 Proc. SMP

- ▶ Processors start by running a loop that sums their subset of vector  $A$  numbers
- ▶ Vectors  $A$  and  $sum$  are **shared** variables
- ▶  $P_n$  is the processor's number,  $i$  is a private variable

```
sum[Pn] = 0;
for (i=1000*Pn; i<1000*(Pn+1); i=i+1)
{
    sum[Pn] = sum[Pn] + A[i];
}
```



# Summing 100,000 Numbers on 100 Proc. SMP

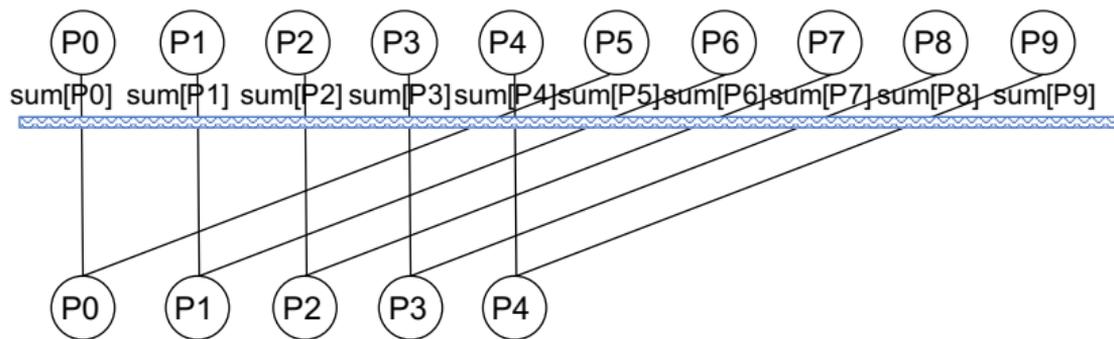
- ▶ The processors then coordinate in adding together the partial sums
- ▶ `half` is a **private** variable initialized to 100 (the number of processors))

```
repeat
    synch();           //synchronize first
    if (half%2 != 0 && Pn == 0) {
        sum[0] = sum[0] + sum[half-1];
    }
    half = half/2
    if (Pn<half) {
        sum[Pn] = sum[Pn] + sum[Pn+half]
    }
until (half == 1); //final sum in sum[0]
```



## Synch () Example

- ▶ `synch ()`: Processors must synchronize before the “consumer” processor tries to read the results from the memory location written by the “producer” processor
- ▶ **Barrier synchronization**: a synchronization scheme where processors wait at the barrier, not proceeding until every processor has reached it



# Barrier Implemented with Spin-Locks

- ▶ `n` is a **shared** variable initialized to the number of processors
- ▶ `count` is a **shared** variable initialized to 0
- ▶ `arrive` and `depart` are **shared** spin-lock variables where `arrive` is initially unlocked and `depart` is initially locked

```
procedure synch()
{
    lock(arrive);
    count = count + 1; // count the processors as
    if (count < n) {   // they arrive at barrier
        unlock(arrive)
    }
    else {
        unlock(depart);
    }
    lock(depart);
    count = count - 1; // count the processors as
    if (count > 0) {  // they leave barrier
        unlock(depart)
    }
    else {
        unlock(arrive);
    }
}
```



# Spin-Locks on Bus Connected ccUMAs

With a bus based cache coherency protocol (write invalidate), spin-locks allow processors to wait on a local copy of the lock in their caches

## Reduces Bus Traffic

Once the processor with the lock releases the lock (writes a 0) all other caches see that write and invalidate their old copy of the lock variable. Unlocking restarts the race to get the lock. The winner gets the bus and writes the lock back to 1. The other caches then invalidate their copy of the lock and on the next lock read fetch the new lock value (1) from memory.

This scheme has problems scaling up to many processors because of the communication traffic when the lock is released and contested



# Message Passing Multiprocessors (MPP)

Each processor has its own private address space

**Q1: How do they share data?**

Processors share data by **explicitly** sending and receiving information (**message passing**)

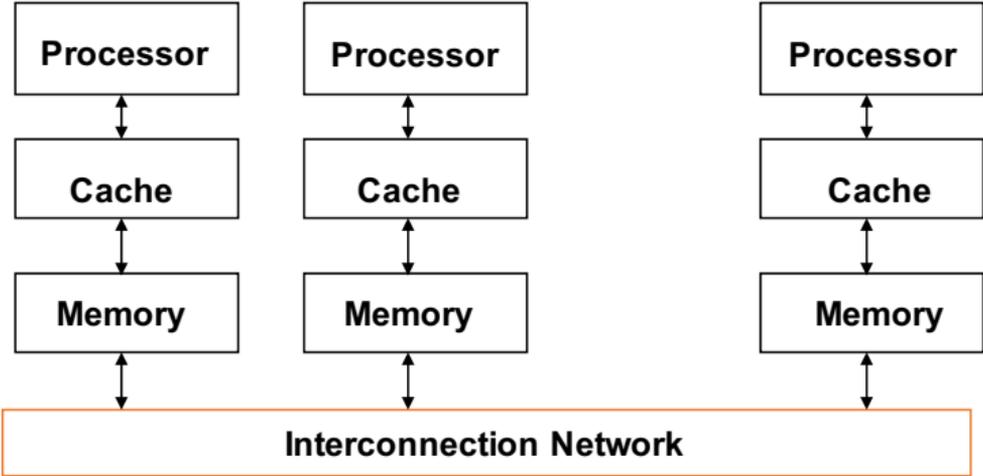
**Q2: How do they coordinate?**

Coordination is built into message passing primitives (**message send** and **message receive**)



# Message Passing Multiprocessors (MPP)

Each processor has its own private address space



# Summing 100,000 Numbers on 100 Proc. MPP

Start by distributing 1000 elements of vector A to each of the **local** memories and summing each subset in parallel

```
sum = 0;
for (i = 0; i<1000; i = i + 1)
{
    sum = sum + A[i]; // sum local array subset
}
```



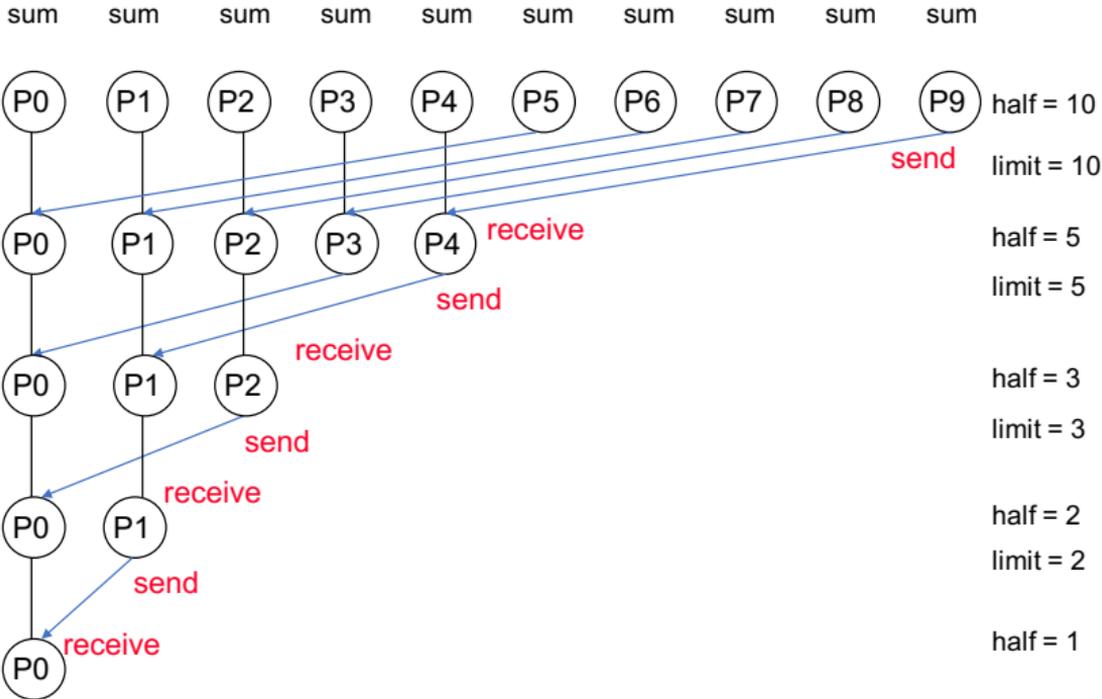
# Summing 100,000 Numbers on 100 Proc. MPP

- ▶ The processors then coordinate in adding together the sub sums
- ▶  $P_n$  is the number of processors
- ▶ `send(x, y)` sends value  $y$  to processor  $x$ , and `receive()` receives a value

```
half = 100;
limit = 100;
repeat {
    half = (half+1)/2; //dividing line
    if (Pn>= half && Pn<limit) send(Pn-half, sum);
    if (Pn<(limit/2)) sum = sum + receive();
    limit = half;
} until (half == 1); //final sum in P0's sum
```



# An Example with 10 Processors



# Pros and Cons of Message Passing

- ▶ 😊 Message passing multiprocessors are much easier for hardware designers to design
- ▶ 😊 Don't have to worry about cache coherency for example
- ▶ 😊 The advantage for programmers is that communication is explicit, so there are fewer “performance surprises” than with the implicit communication in cache-coherent SMPs.
  
- ▶ 😞 Message sending and receiving is much slower than addition
- ▶ 😞 Harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance \*.

---

\*With cache-coherent shared memory, the hardware figures out what data needs to be communicated ▶



# Review: Multiprocessor Basics

- ▶ Q1: How do they share data?
- ▶ Q2: How do they coordinate?
- ▶ Q3: How scalable is the architecture? How many processors?

|                     |                 | # of Proc |           |
|---------------------|-----------------|-----------|-----------|
| Communication model | Message passing |           | 8 to 2048 |
|                     | Shared address  | NUMA      | 8 to 256  |
|                     |                 | UMA       | 2 to 64   |
| Physical connection | Network         |           | 8 to 256  |
|                     | Bus             |           | 2 to 36   |

