

Introduction to Parallel Computing

Instructor

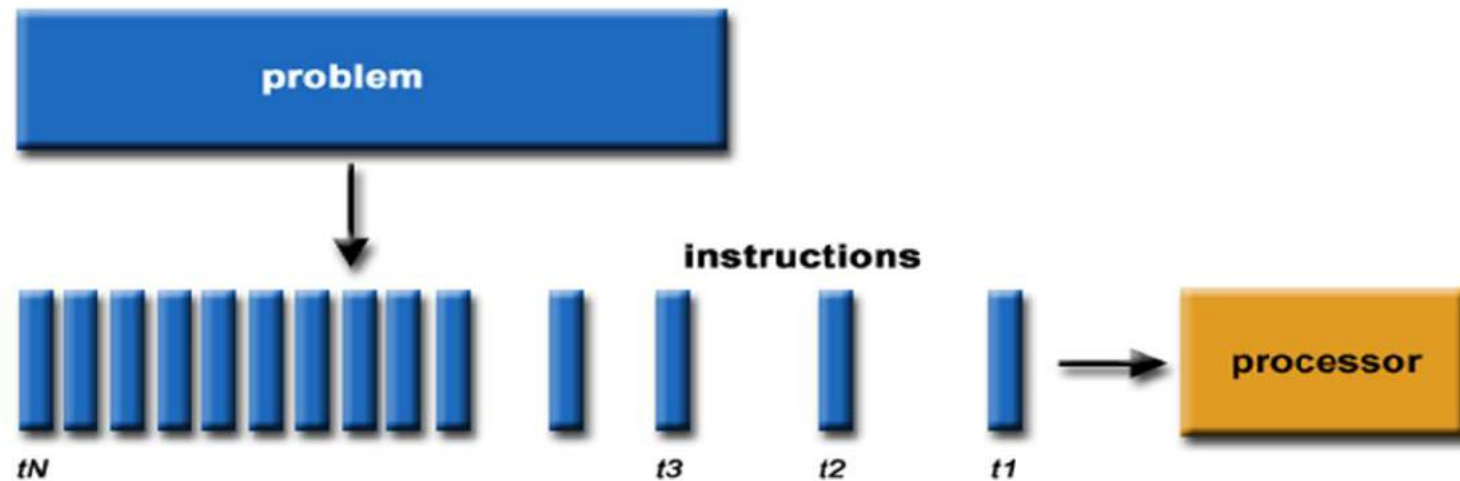
Dr. B Krishna Priya

Contents

- Introduction to Parallel computing
- Motivating Parallelism
- Scope of Parallelism
- Parallel Programming Platforms
- Implicit Parallelism: Trends in Microprocessor Architectures
- Limitations of Memory System Performance
- Dichotomy of Parallel Computing Platforms
- Physical Organization of Parallel Platforms

What is parallel computing

- Traditional software has been written for serial computing.
- To be run on single computer with single CPU

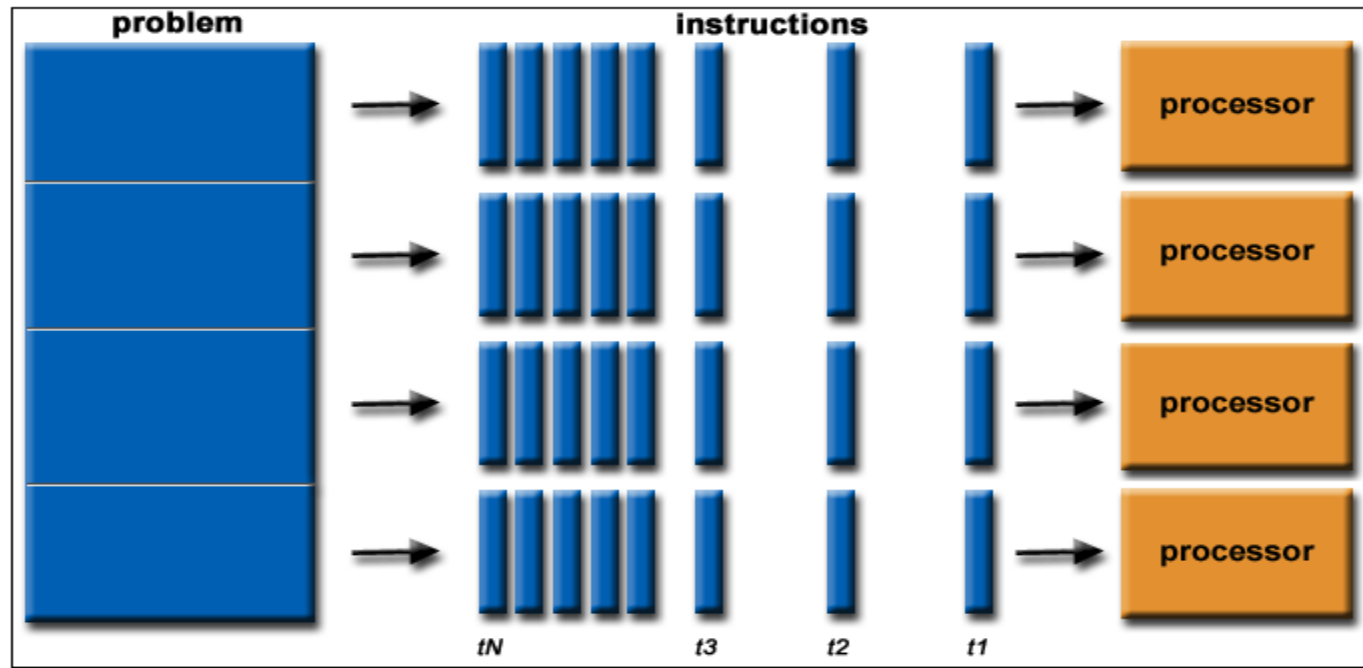


Limitations of Serial Computing

- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever faster serial computers.
- Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware.
- Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
- Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

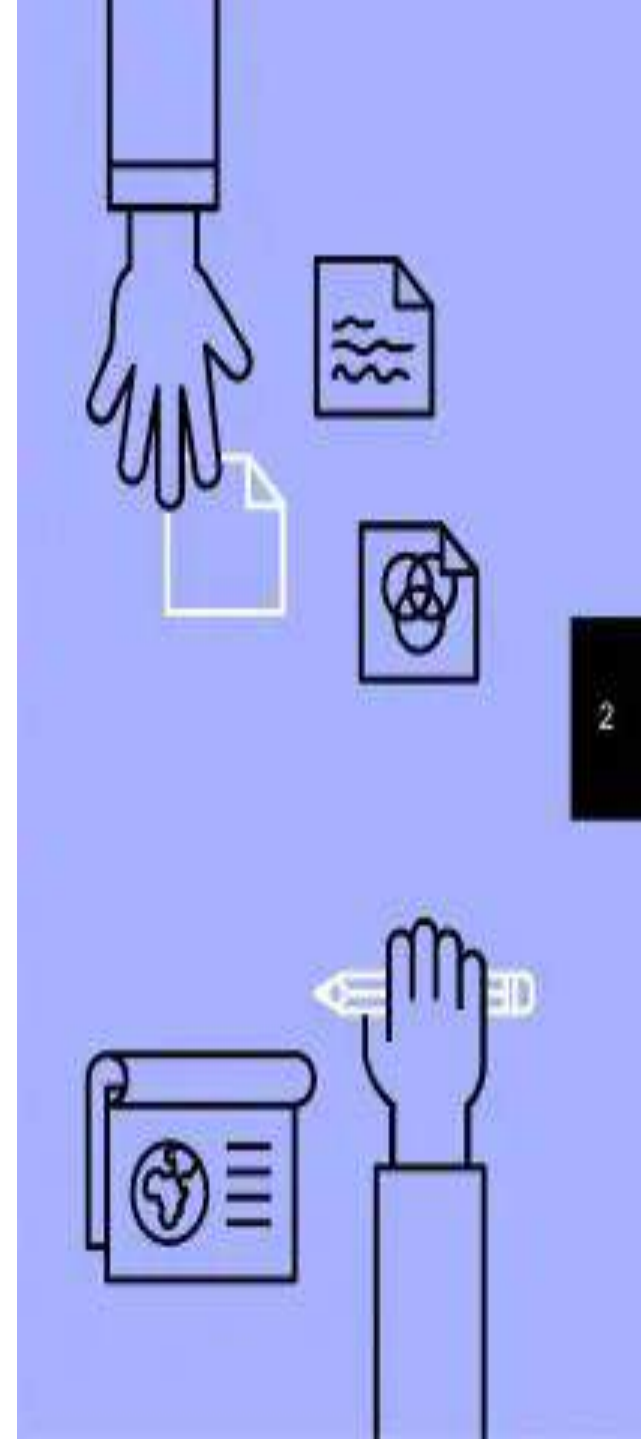
What is parallel computing

- In the simplest sense, parallel computing is the simultaneously use of multiple computing resource to solve computational problem.



Parallel computing

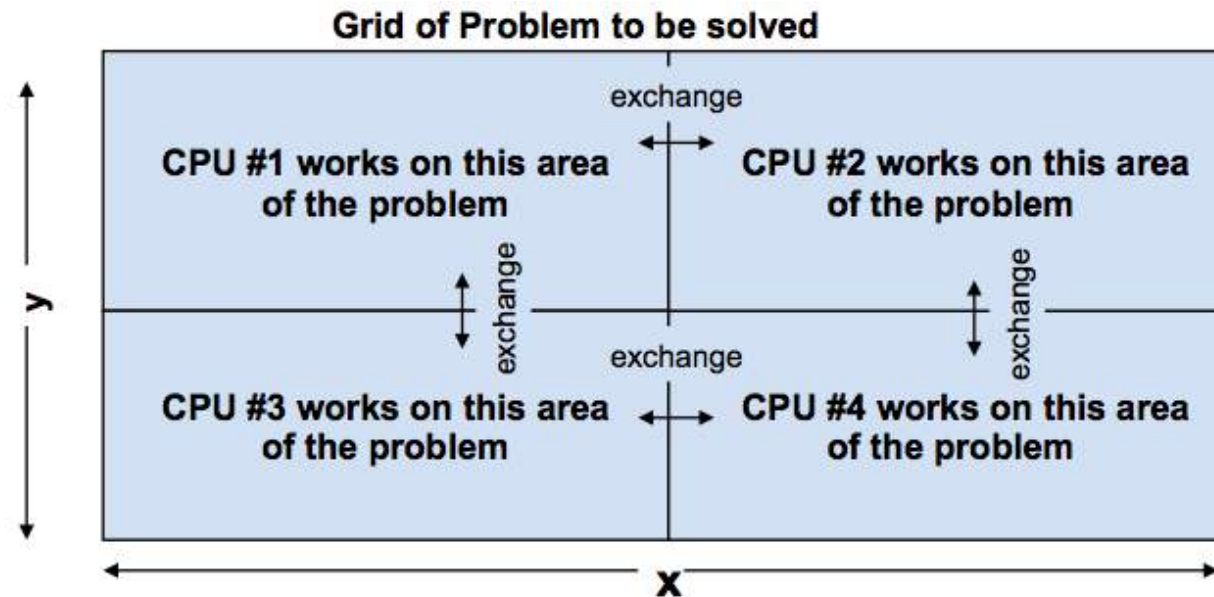
- A parallel computing is a “Collection of processing elements that co-operate and communicate to solve large Problem”
- Basic Principle: A computation can be divided into smaller subproblem each of which can be solved simultaneously.
- Assumption: Parallel computing assumes that a parallel hardware is present at our disposal, which is capable of executing these computations in parallel.



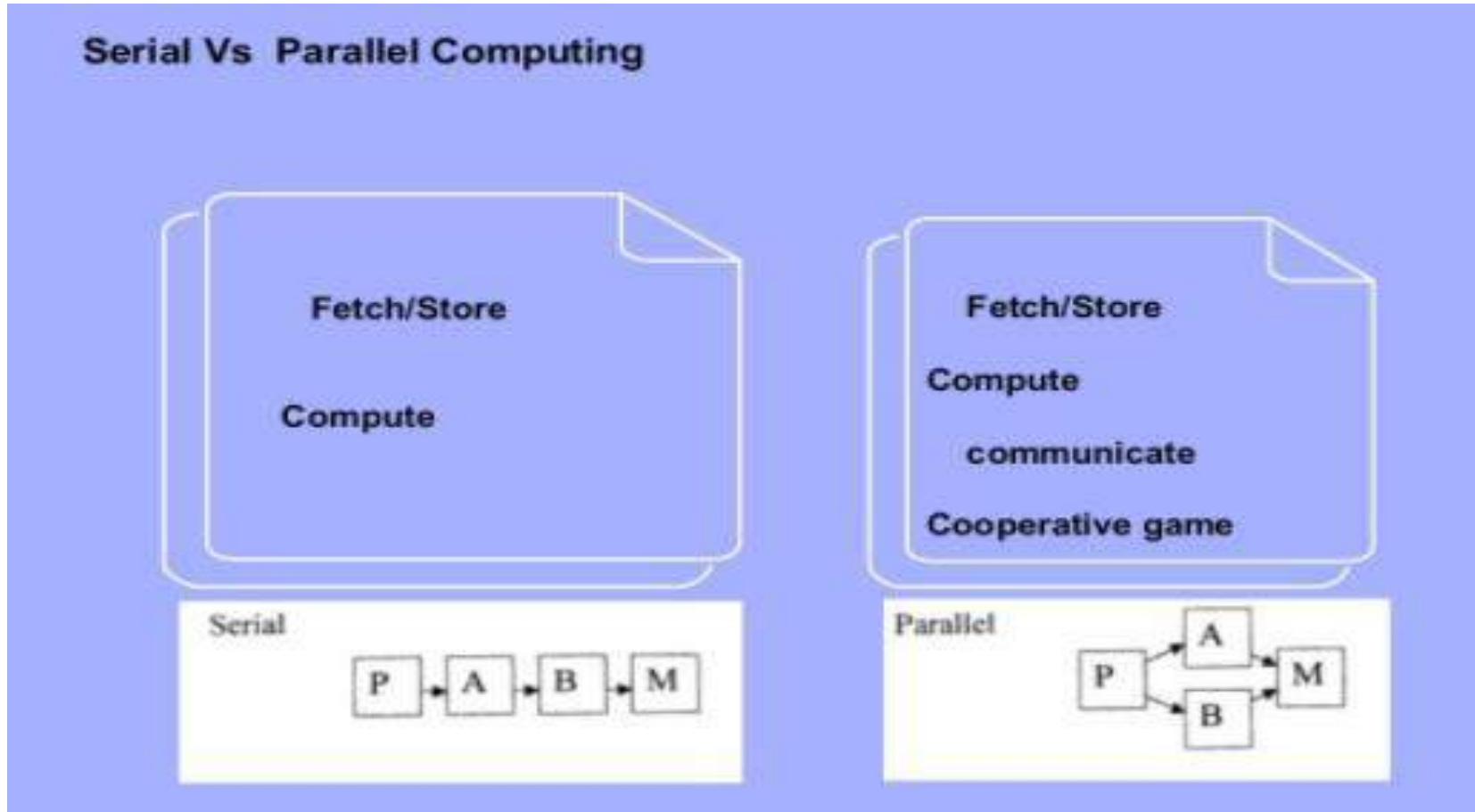
The concept of parallel computing

- One processor in your desktop or laptop can give you output in X hours
- $X > 10$
- Why not use several (Y) processors, and get the results in $X < 1$?

- Each processor works on one part of the problem
- Processors can exchange information with other processors



Serial vs Parallel Computing



Motivating Parallel Computing

The primary reasons for using parallel computing:

- Save time - wall clock time
- Solve larger problems
- Provide concurrency (do multiple things at the same time)

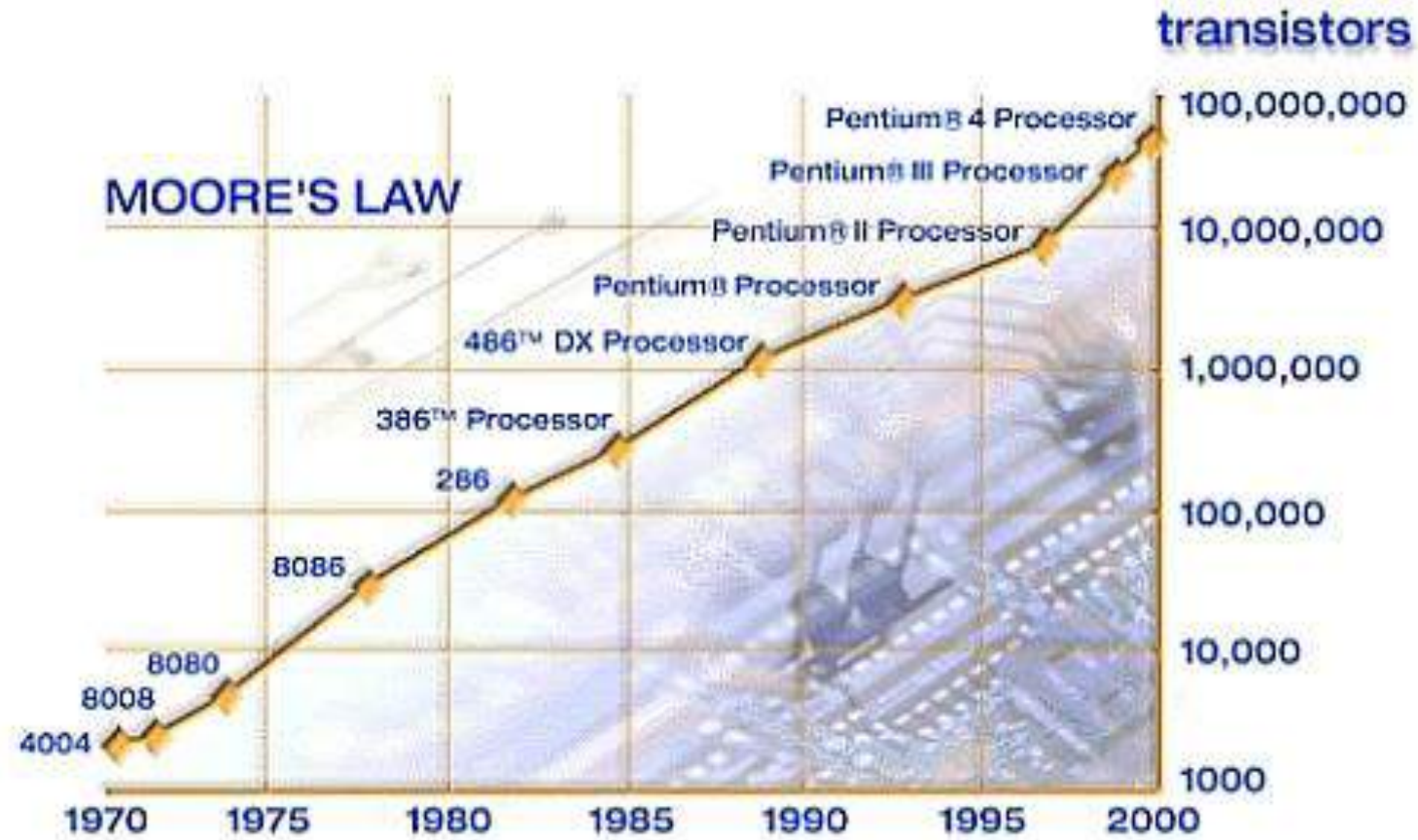
Motivating Parallel Computing

Computational Power argument

Moore's law states [1965]:

- Moore law is a prediction formulated by Moore in 1965 about transistor density on integrated circuit chip(ICs).\
- According to Moore's Law, “ the number of transistors used per square inch in the integrated circuits doubles about every 2 years.”
- As a result the scale gets smaller and smaller and performance of microprocessor has enjoyed an exponential growth.

Technology Growth Graph



Motivating Parallel Computing

The Memory/Disk Speed Argument

- While clock rates of high-end processors have increased at roughly 40% per year over the past decade, DRAM access times have only improved at the rate of roughly 10% per year over this interval.
- This mismatch in speeds causes significant performance bottlenecks.
- Parallel platforms provide increased bandwidth to the memory system.
- Parallel platforms also provide higher aggregate caches.
- Principles of locality of data reference and bulk access, which guide parallel algorithm design also apply to memory optimization.
- Some of the fastest growing applications of parallel computing utilize not their raw computational speed, rather their ability to pump data to memory and disk faster.

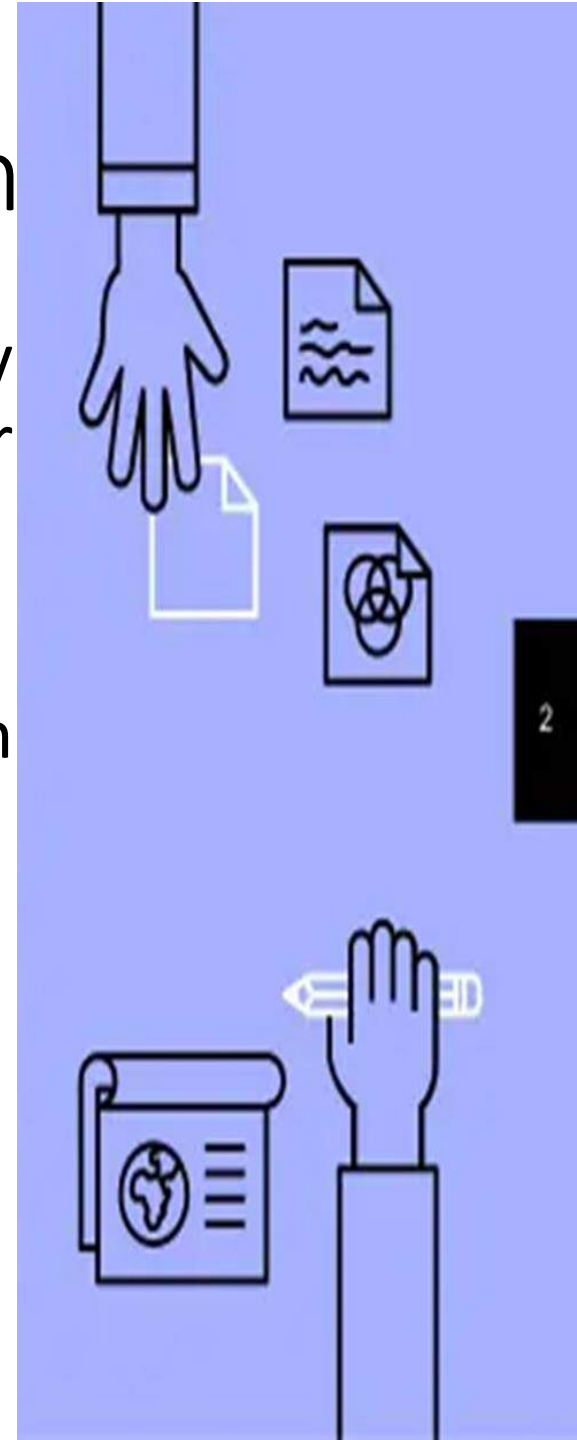
Motivating Parallel Computing

The Data Communication Argument

- As the network evolves, the vision of the Internet as one large computing platform has emerged.
- In many other applications (typically databases and data mining) the volume of data is such that they cannot be moved.
- Any analyses on this data must be performed over the network using parallel techniques.

Scope of Parallel computin

- Parallelism finds applications in very diverse application domains for different motivating reasons.
- These range from improved application performance to cost considerations.



Applications

- Commercial computing
 - Weather forecasting
 - Remote sensors, image processing
 - Process optimization, operation resea
- Scientific and Engineering applicati
 - Computational chemistry
 - Molecular modeling
 - Structure mechanics
- Business application
 - E-governance
 - Medical imaging
- Internet applications
 - Internet servers
 - Digital libraries





Performance



What is the Performance?

Plane	DC to Paris	Speed	Passengers	passengers X mph
Boeing 747	6.5 hours	610 mph	470	286,700
Concorde	3 hours	1350 mph	132	178,200

Which of the planes has better performance

- The plane with the highest speed is **Concorde**
- The plane with the largest capacity is **Boeing 747**

Performance Example

- Time of Concorde vs. Boeing 747?
 - Concorde is 1350 mph / 610 mph = 2.2 times faster
- Throughput of Concorde vs. Boeing 747 ?
 - Boeing is 286,700 pmph / 178,200 pmph = 1.6 times faster
- Boeing is 1.6 times faster in terms of throughput
- Concorde is 2.2 times faster in terms of flying time
- When discussing processor performance, we will focus primarily on execution time for a single job - why?

Definitions of Time

- Time can be defined in different ways, depending on what we are measuring:
 - **Response time** : The time between the start and completion of a task. It includes time spent executing on the CPU, accessing disk and memory, waiting for I/O and other processes, and operating system overhead. This is also referred to as **execution time**.
 - **Throughput** : The total amount of work done in a given time.
 - **CPU execution time** : Total time a CPU spends computing on a given task (excludes time for I/O or running other programs). This is also referred to as simply **CPU time**.
-

Performance Definition

- For some program running on machine X,
Performance = $1 / \text{Execution time}_X$
- "X is n times faster than Y"
Performance_X / Performance_Y = n

Problem:

- machine A runs a program in 20 seconds
- machine B runs the same program in 25 seconds
- how many times faster is machine A?

$$\frac{25}{20} = 1.25$$

Basic Measurement Metrics

- Comparing Machines

- Metrics

- Execution time
 - Throughput
 - CPU time

Computer Clock

- A **computer clock** runs at a constant rate and determines when events take place in hardware.

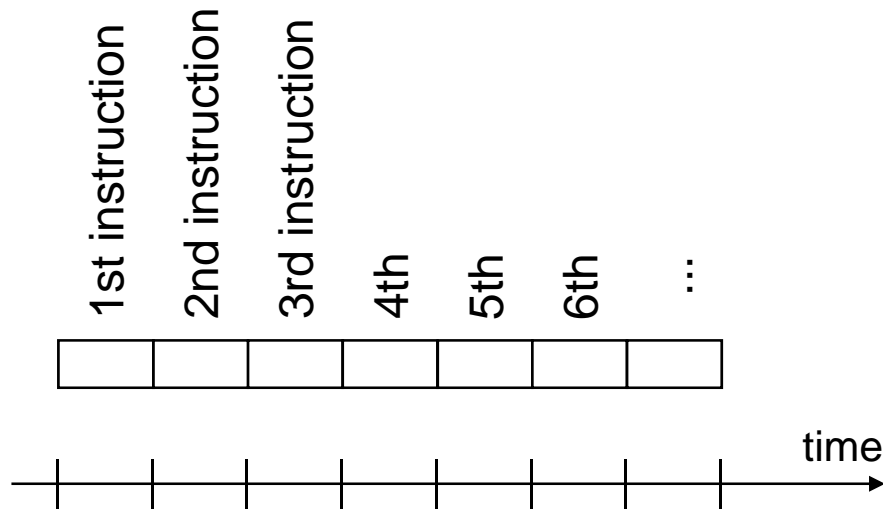


- The **clock cycle time** is the amount of time for one **clock period** to elapse (e.g. 5 ns).
- The **clock rate** is the inverse of the **clock cycle time**.
- For example, if a computer has a **clock cycle time** of 5 ns, the **clock rate** is:

$$\frac{1}{5 \times 10^{-9} \text{ sec}} = 200 \text{ MHz}$$

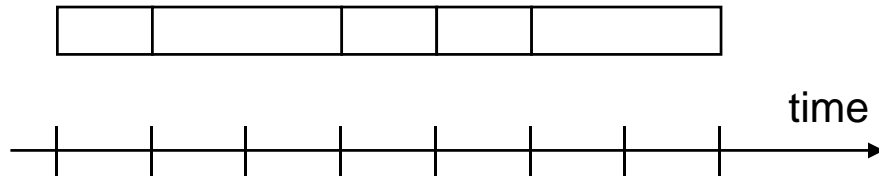
How Many Cycles are Required for a Program?

- Could assume that # of cycles = # of instructions



- This assumption is incorrect, different instructions take different amounts of time on different machines.

Different Numbers of Cycles for Different Instructions



- Division takes more time than addition
- Floating point operations take longer than integer ones
- Accessing memory takes more time than accessing registers

Now That We Understand Cycles

- A given program will require
 - some number of instructions (machine instructions)
 - some number of clock cycles
 - some number of seconds
 - We have a vocabulary that relates these quantities:
 - clock cycle time (seconds per cycle)
 - clock rate (cycles per second)
 - CPI (cycles per instruction)
 - *a floating point intensive application might have a higher CPI*
-

Computing CPU Time

- The time to execute a given program can be computed as
$$\text{CPU time} = \text{CPU clock cycles} \times \text{clock cycle time}$$
- Since clock cycle time and clock rate are reciprocals
$$\text{CPU time} = \text{CPU clock cycles} / \text{clock rate}$$
- The number of CPU clock cycles can be determined by
$$\begin{aligned}\text{CPU clock cycles} &= (\text{instructions/program}) \times (\text{clock cycles/instruction}) \\ &= \text{Instruction count} \times \text{CPI}\end{aligned}$$

which gives

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} \times \text{clock cycle time}$$

$$\text{CPU time} = \text{Instruction count} \times \text{CPI} / \text{clock rate}$$

- The units for CPU time are

$$\text{CPU time} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{clock cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{clock cycle}}$$

Which factors are affected by each of the following?

	instr. Count	CPI	clock rate
Program	x		
Compiler	x	x	
Instr. Set Arch.	x	x	
Organization		x	x
Technology			x

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

CPU Time Example

■ Example 1:

- ❑ CPU clock rate is 1 MHz
- ❑ Program takes 45 million cycles to execute
- ❑ What's the CPU time?

$$45,000,000 * (1 / 1,000,000) = 45 \text{ seconds}$$

■ Example 2:

- CPU clock rate is 500 MHz
- Program takes 45 million cycles to execute
- What's the CPU time

$$45,000,000 * (1 / 500,000,000) = 0.09 \text{ seconds}$$

CPI Example

- **Example:** Let assume that a benchmark has 100 instructions:
 - 25 instructions are loads/stores (each take 2 cycles)
 - 50 instructions are adds (each takes 1 cycle)
 - 25 instructions are square root (each takes 50 cycles)

What is the CPI for this benchmark?

$$\text{CPI} = ((0.25 * 2) + (0.50 * 1) + (0.25 * 50)) = 13.5$$

Computing CPI

- The CPI is the average number of cycles per instruction.
- If for each instruction type, we know its frequency and number of cycles need to execute it, we can compute the overall CPI as follows:

$$\text{CPI} = \sum \text{CPI} \times F$$

- For example

Op	F	CPI	CPI x F	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%
Total	100%		2.2	100%

Performance

- Performance is determined by execution time
- Do you think any of the variables is sufficient enough to determine computer performance?
 - ❑ # of cycles to execute program?
 - ❑ # of instructions in program?
 - ❑ # of cycles per second?
 - ❑ average # of cycles per instruction?
 - ❑ average # of instructions per second
- It is not true to think that one of the variables is indicative of performance.

CPI Example

- Suppose we have two implementations of the same instruction set architecture (ISA).

For some program,

Machine A has a clock cycle time of **10 ns.** and a CPI of **2.0**

Machine B has a clock cycle time of **20 ns.** and a CPI of **1.2**

- Which machine is faster for this program, and by how much?

Assume that # of instructions in the program is 1,000,000,000.

$$\text{CPU Time}_A = 10^9 * 2.0 * 10 * 10^{-9} = 20 \text{ seconds}$$

$$\text{CPU Time}_B = 10^9 * 1.2 * 20 * 10^{-9} = 24 \text{ seconds}$$

Machine A is faster

$$\frac{24}{20} = 1.2 \text{ times}$$

Number of Instruction Example

- A compiler designer is trying to decide between two code sequences for a particular machine. Based on the hardware implementation, there are three different classes of instructions: Class A, Class B, and Class C, and they require one, two, and three cycles (respectively).

The first code sequence has 5 instructions: 2 of A, 1 of B, and 2 of C

The second sequence has 6 instructions: 4 of A, 1 of B, and 1 of C.

- Which sequence will be faster? How much?
- What is the CPI for each sequence?

of cycles for first code = $(2 * 1) + (1 * 2) + (2 * 3) = 10$ cycles


of cycles for second code = $(4 * 1) + (1 * 2) + (1 * 3) = 9$ cycles

$$10 / 9 = 1.11 \text{ times}$$


$$\text{CPI for first code} = 10 / 5 = 2$$

$$\text{CPI for second code} = 9 / 6 = 1.5$$

Parallel Programming Platforms

- ▶ In traditional view of a computer, processor and memory are connected by data path.
 - ▶ Processor, memory and datapath present an bottlenecks to the overall processing rate of a computer system.
 - ▶ To overcome, multiplicity is implemented in processing unit, memory and datapath, which is shown either directly in implicit architecture and indirectly to the programmer in different forms.
 - ▶ The main objective is to provide sufficient details to programmer to be able to write efficient code on variety of platform.
 - ▶ Performance of various parallel algorithm.
- 

Implicit Parallelism: Trends in Microprocessor Architectures*

- ▶ Microprocessor clock speeds have posted impressive gains over the past two decades (two to three orders of magnitude).
 - ▶ Higher levels of device integration have made available a large number of transistors.
 - ▶ The question of how best to utilize these resources is an important one.
 - ▶ Current processors use these resources in multiple functional units and execute multiple instructions in the same cycle.
 - ▶ The precise manner in which these instructions are selected and executed provides impressive diversity in architectures.
- 

Pipelining and Superscalar Execution

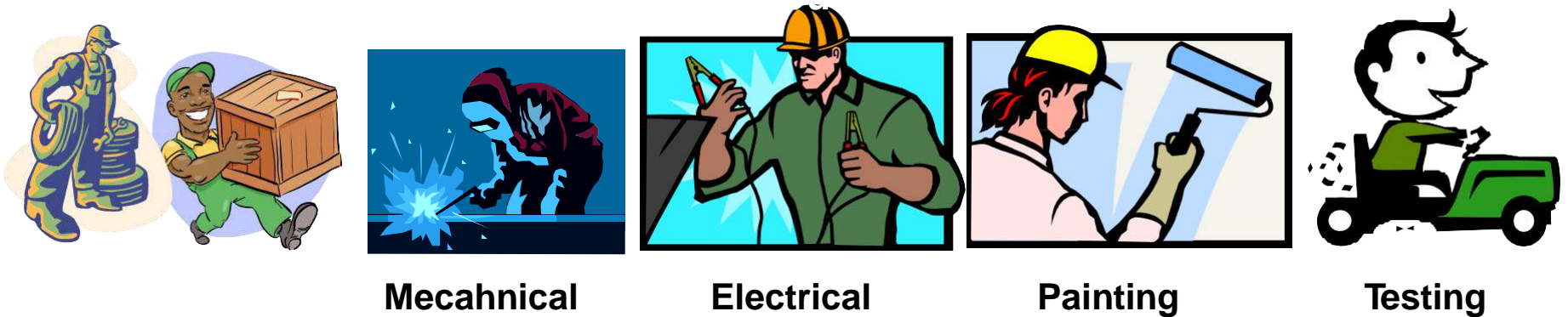
- ▶ Pipelining overlaps various stages of instruction execution to achieve performance.
- ▶ At a high level of abstraction, an instruction can be executed while the next one is being decoded and the next one is being fetched.
- ▶ This is akin to an assembly line for manufacture of cars.

Automobile Team Assembly



car assembled every four hours
6 cars per day
180 cars per month
2,040 cars per year

Automobile Assembly Line



First car assembled in 4 hours (pipeline latency)
thereafter, 1 car completed per hour
21 cars on first day, thereafter 24 cars per day
717 cars per month
8,637 cars per year
What gives 4X increase?

Throughput: Team Assembly

Red car
started

Red car
completed

Mechanical Electrical Painting Testing

Mechanical Electrical Painting Testing

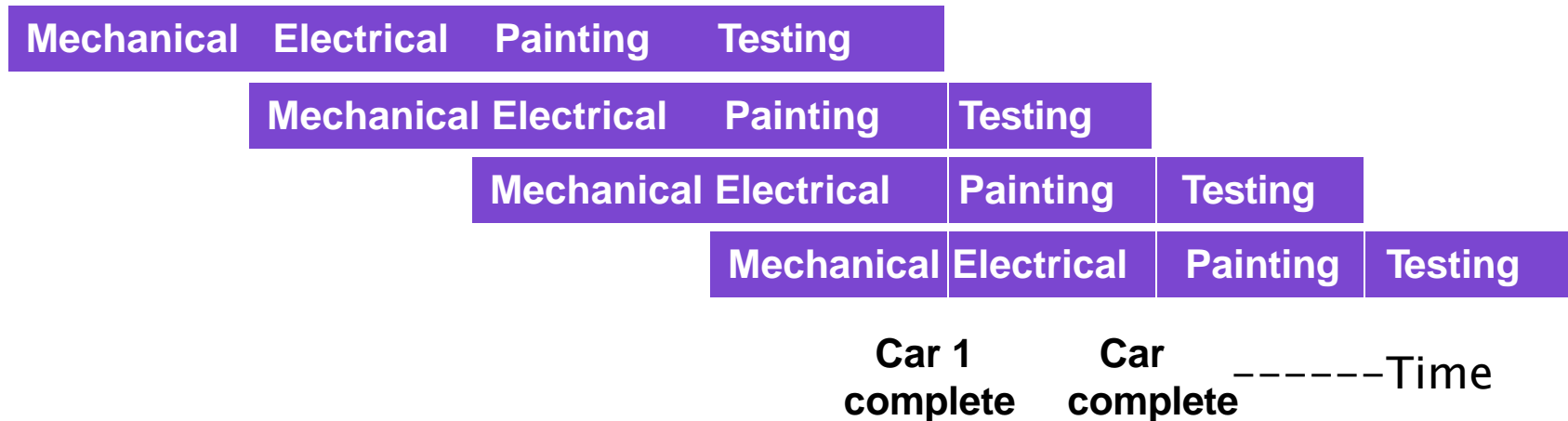
Blue car
started

Blue car
completed

Time of assembling one car = n hours where n is the number of nearly equal subtasks, each requiring 1 unit of time

Throughput = $1/n$ cars per unit time

Throughput: Assembly Line



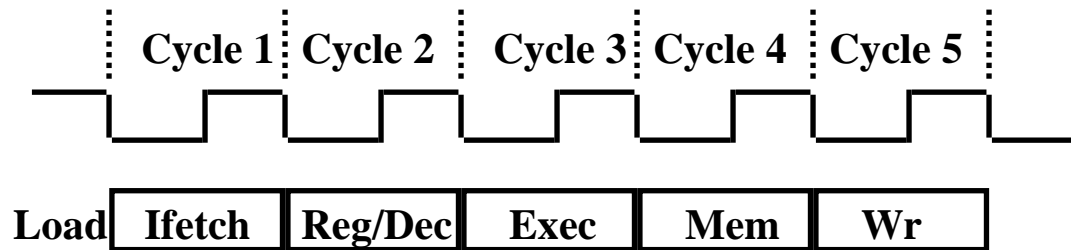
Time to complete first car = n time units (latency)

Cars completed in time $T = T - n + 1$

Throughput = $1 - (n - 1) / T$ cars per unit time

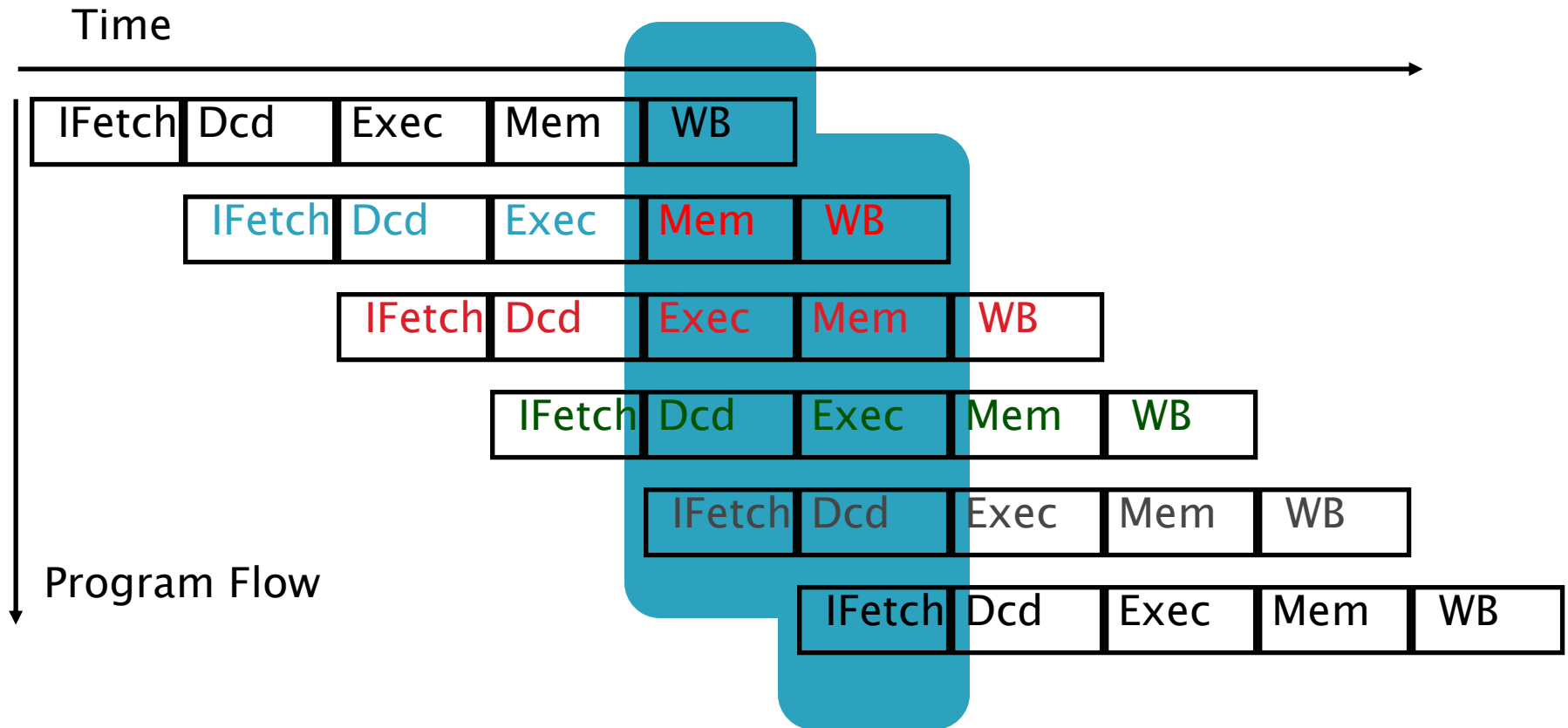
$$\frac{\text{Throughput (assembly line)}}{\text{Throughput (team assembly)}} = \frac{1 - (n - 1) / T}{1/n} = n - \frac{n(n - 1)}{T} \rightarrow n \text{ as } T \rightarrow \infty$$

Five Stages of an Instruction

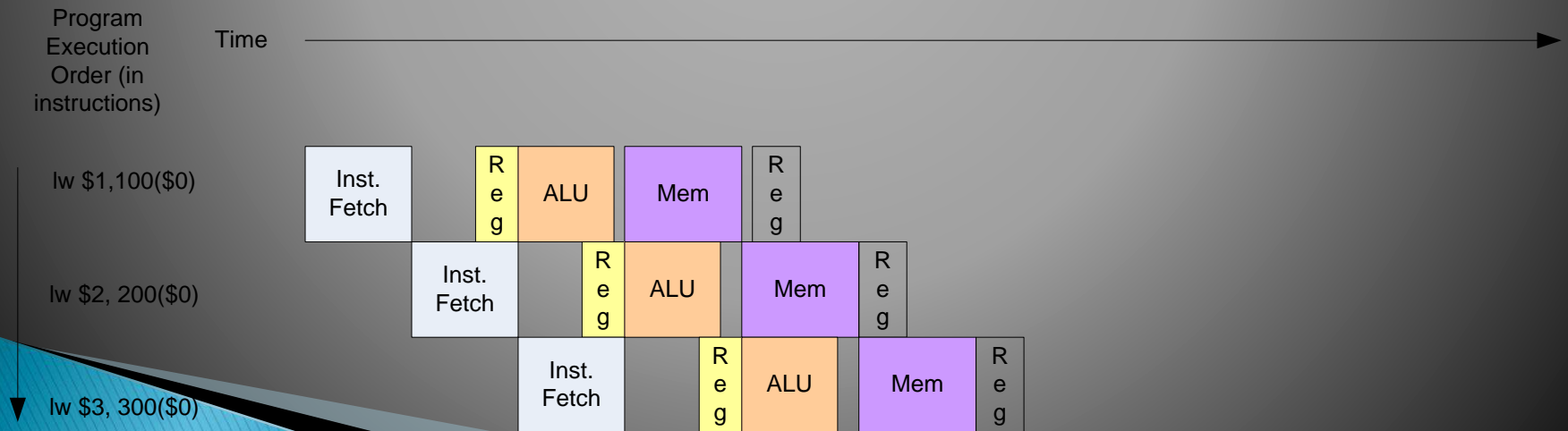
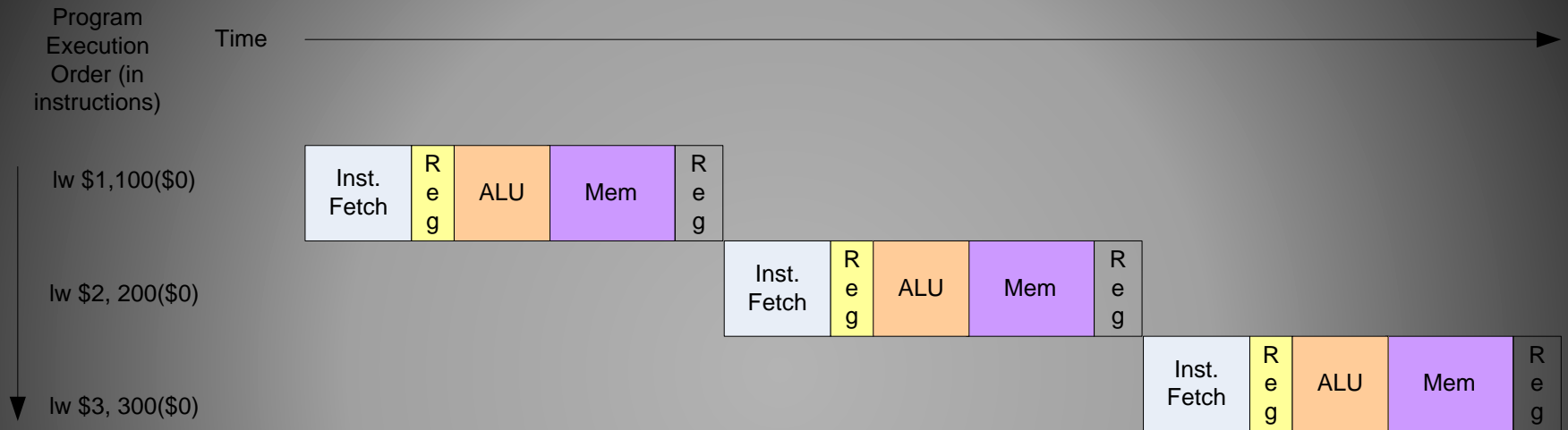


- ▶ **Ifetch: Instruction Fetch**
 - Fetch the instruction from the Instruction Memory
- ▶ **Reg/Dec: Registers Fetch and Instruction Decode**
- ▶ **Exec: Calculate the memory address**
- ▶ **Mem: Read the data from the Data Memory**
- ▶ **Wr: Write the data back to the register file**

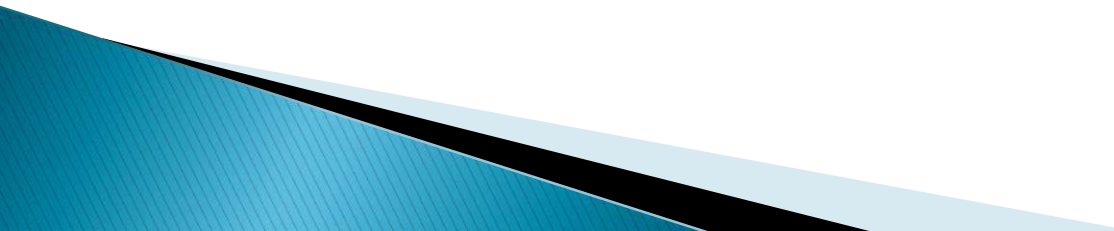
Conventional Pipelined Execution Representation



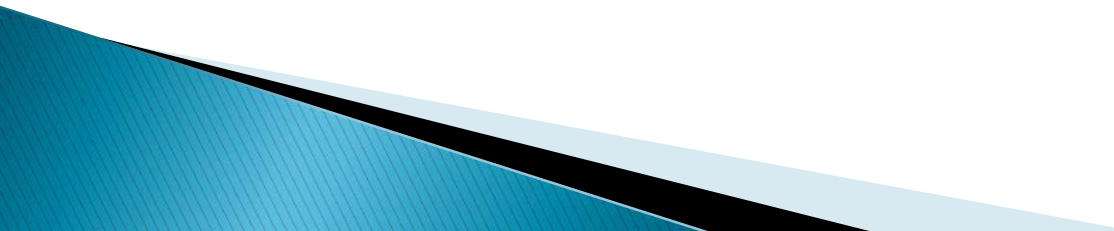
Example



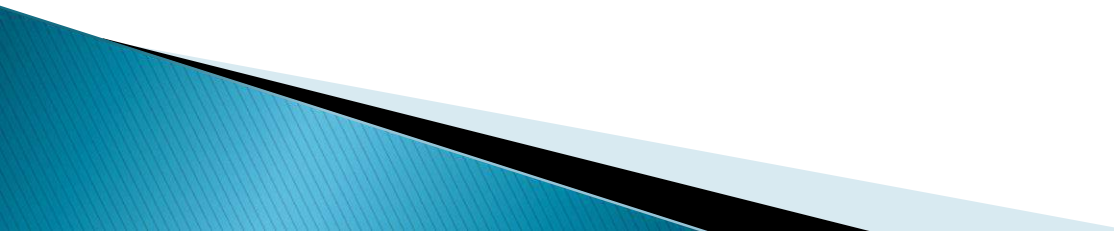
Pipelining and Superscalar Execution

- ▶ Processor have relied on pipelines for improving execution rate.
 - ▶ To increase speed of a single pipeline, one would break down the tasks into smaller units.
 - ▶ To improve execution rate is to use multiple pipelines.
- 

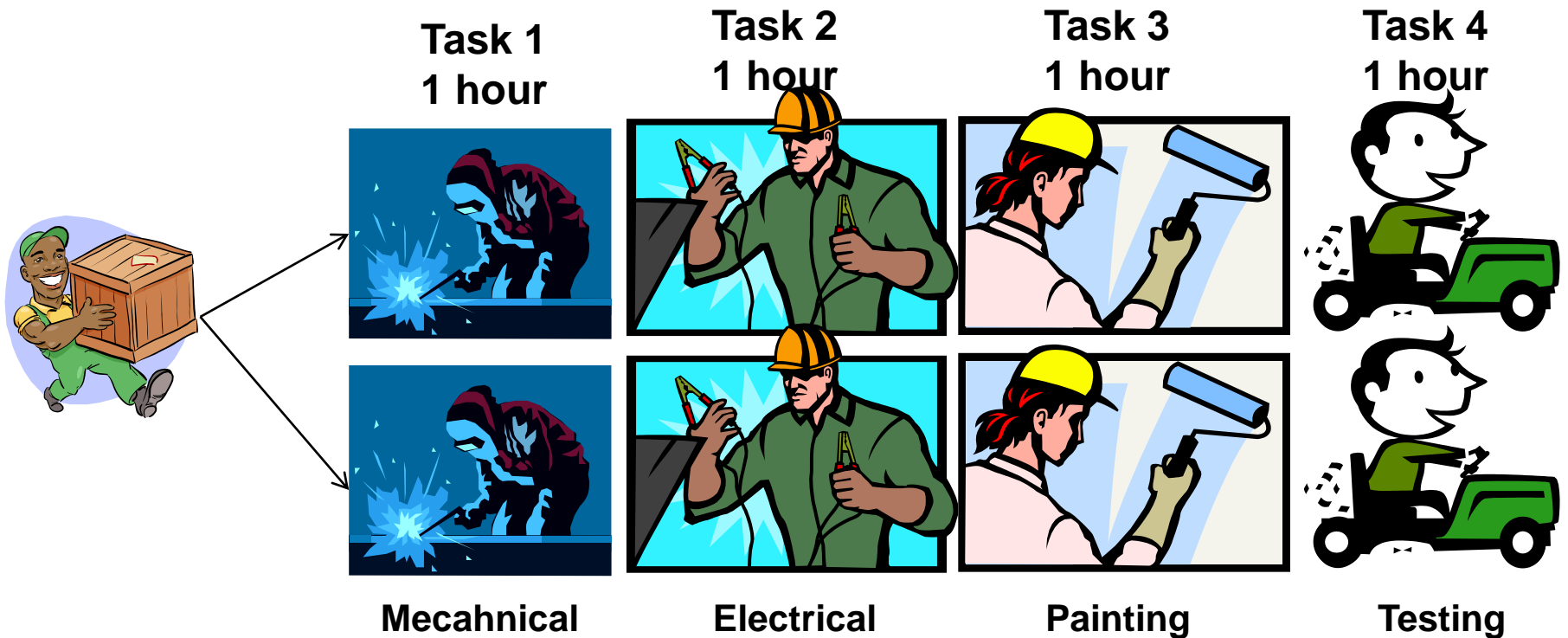
Pipelining and Superscalar Execution

- ▶ Pipelining, however, has several limitations.
 - ▶ The speed of a pipeline is eventually limited by the slowest stage.
 - ▶ For this reason, conventional processors rely on very deep pipelines (20 stage pipelines in state-of-the-art Pentium processors).
 - ▶ However, in typical program traces, every 5- 6th instruction is a conditional jump! This requires very accurate branch prediction.
 - ▶ The penalty of a misprediction grows with the depth of the pipeline, since a larger number of instructions will have to be flushed.
- 

Pipelining and Superscalar Execution

- ▶ One simple way of alleviating these bottlenecks is to use multiple pipelines. The question then becomes one of selecting these instructions.
 - ▶ Ability of processor to issue multiple instruction in the same cycle is – superscalar execution.
- 

Multiple Assembly Line



Two cars are assembled in 4 hours (pipeline latency)
thereafter 2 cars per hour
42 cars on the first day and thereafter 48 cars per day
1,432 cars per month
17,272 cars per year

Throughput: Multiple Assembly Line

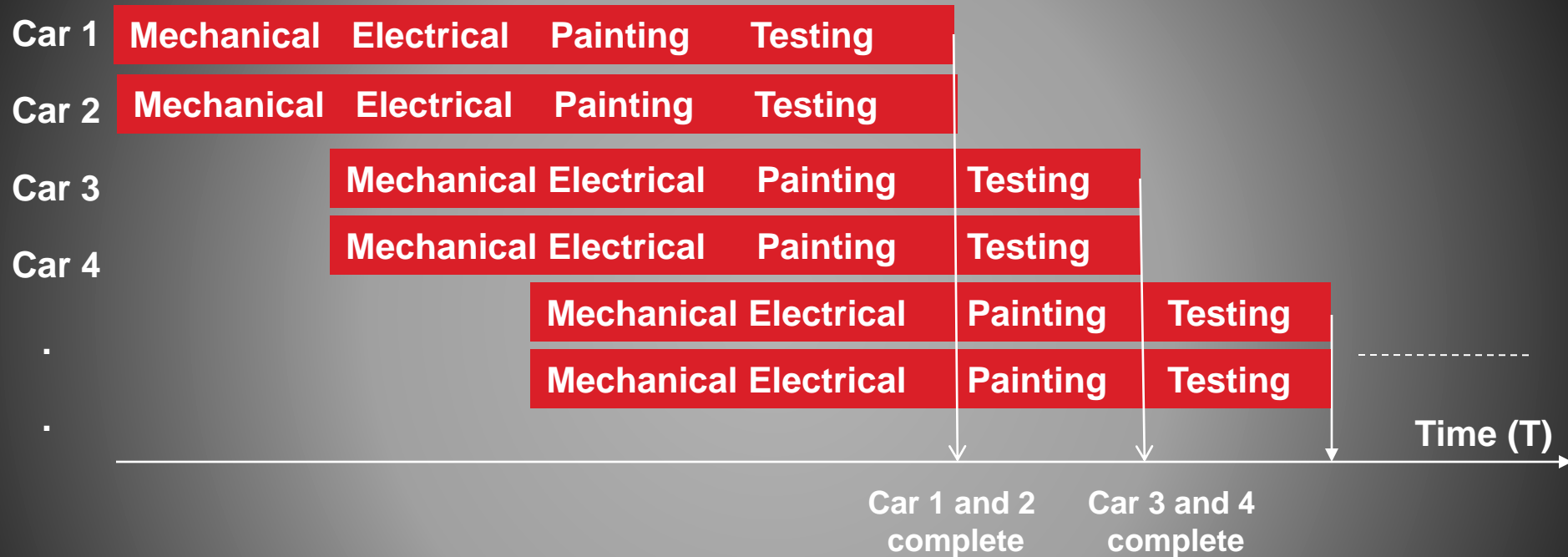
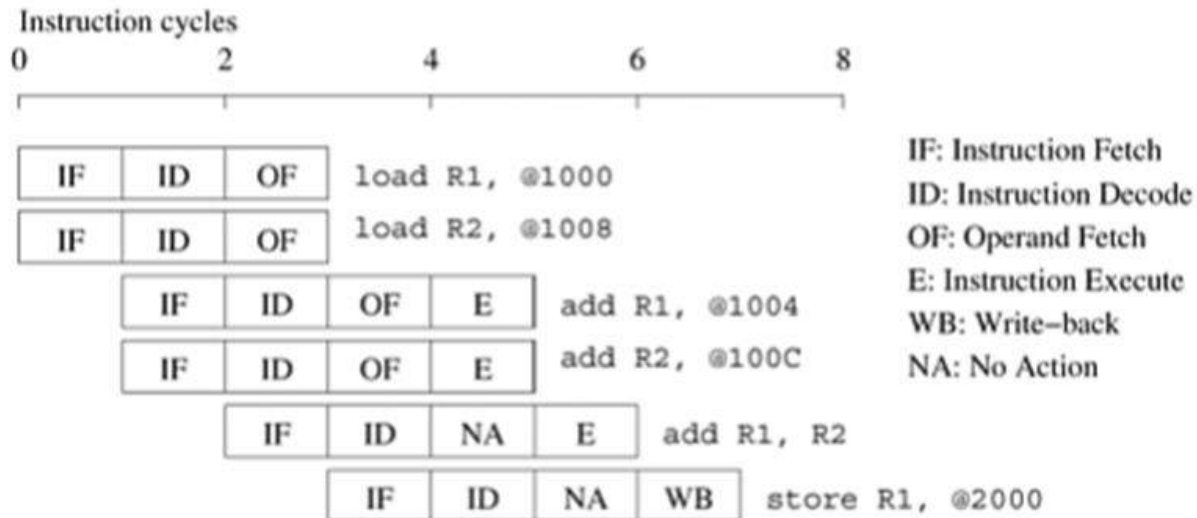


Figure 2.1. Example of a two-way superscalar execution of instructions.

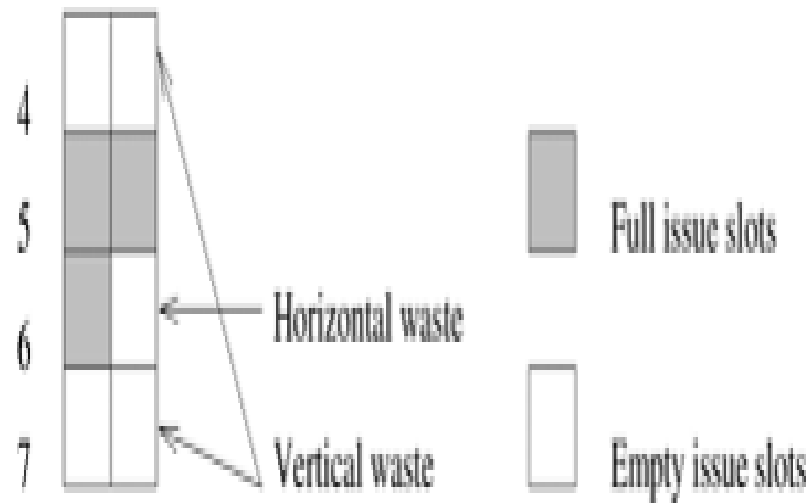
1. load R1, @1000	1. load R1, @1000	1. load R1, @1000
2. load R2, @1008	2. add R1, @1004	2. add R1, @1004
3. add R1, @1004	3. add R1, @1008	3. load R2, @1008
4. add R2, @100C	4. add R1, @100C	4. add R2, @100C
5. add R1, R2	5. store R1, @2000	5. add R1, R2
6. store R1, @2000		6. store R1, @2000
(i)	(ii)	(iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.

Clock cycle

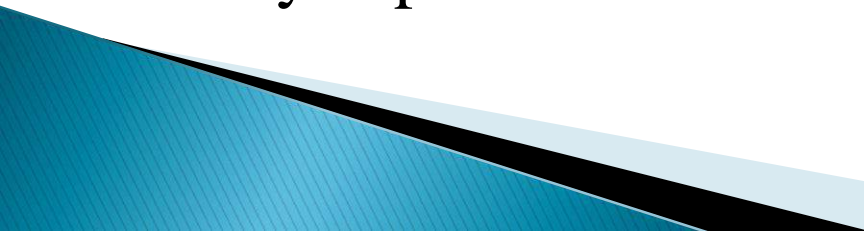


Adder Utilization

(c) Hardware utilization trace for schedule in (b).

Superscalar Execution

Issue mechanism:

- ▶ Not all functional units can be kept busy at all times.
 - ▶ If during a cycle, no functional units are utilized, this is referred to as vertical waste.
 - ▶ If during a cycle, only some of the functional units are utilized, this is referred to as horizontal waste.
 - ▶ Due to limited parallelism in typical instruction traces, dependencies, or the inability of the scheduler to extract parallelism, the performance of superscalar processors is eventually limited.
 - ▶ Conventional microprocessors typically support four-way superscalar execution.
- 


Superscalar Execution

Issue mechanism:

- ▶ In the above example, there is some wastage of resources due to data dependencies.
- ▶ The example also illustrates that different instruction mixes with identical semantics can take significantly different execution time.

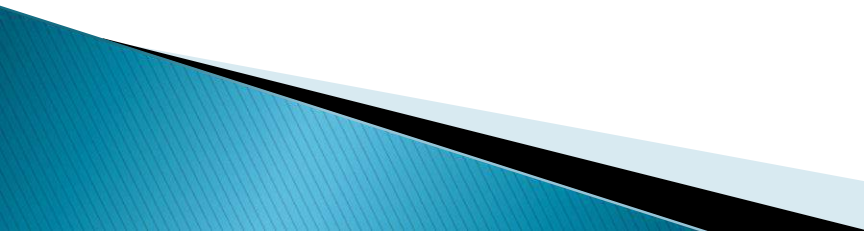
Superscalar Execution

Issue mechanism:

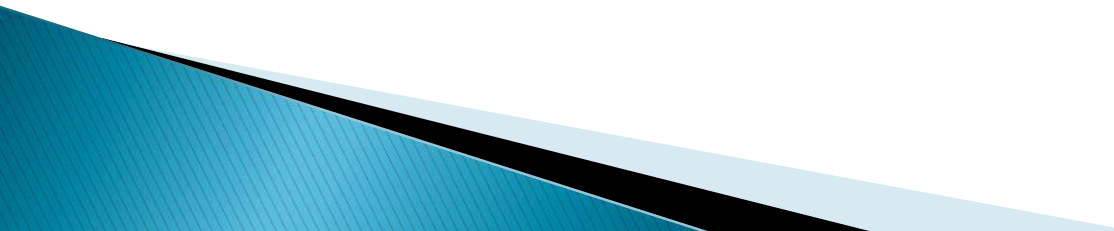
- ▶ Scheduling of instructions is determined by a number of factors:
 - True Data Dependency: The result of one operation is an input to the next.
 - Resource Dependency: Two operations require the same resource.
 - Branch Dependency: For conditional branch instruction destination is known only at the point of execution, scheduling instruction prior may lead to error.
 - The scheduler, a piece of hardware looks at a large number of instructions in an instruction queue and selects appropriate number of instructions to execute concurrently based on these factors.
 - The complexity of this hardware is an important constraint on superscalar processors.
- 

Superscalar Execution

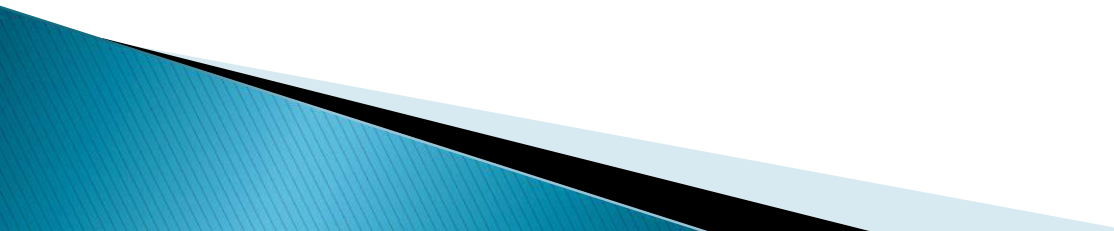
Issue mechanism:

- ▶ In the simpler model, instructions can be issued only in the order in which they are encountered. That is, if the second instruction cannot be issued because it has a data dependency with the first, only one instruction is issued in the cycle. This is called in-order issue.
 - ▶ In a more aggressive model, instructions can be issued out of order. In this case, if the second instruction has data dependencies with the first, but the third instruction does not, the first and third instructions can be co-scheduled. This is also called dynamic issue.
 - ▶ Performance of in-order issue is generally limited.
- 


Very Long Instruction Word (VLIW)Processor

- ▶ The hardware cost and complexity of the superscalar scheduler is a major consideration in processor design.
 - ▶ To address this issues, VLIW processors rely on compile time analysis to identify and bundle together instructions that can be executed concurrently.
 - ▶ These instructions are packed and dispatched together as single instruction long word, and thus the name very long instruction word.
- 

Very Long Instruction Word (VLIW)Processor:Advantages

- ▶ To resolve dependency and resource availability at compile time
 - Scheduling is done in software
 - Decoding is easy
 - Additional parallel instruction made available to control parallel execution
- 

Very Long Instruction Word (VLIW)Processor: Limitations

- ▶ Issue hardware is simpler.
 - ▶ Compiler has a bigger context from which to select co-scheduled instructions.
 - ▶ Compilers, however, do not have runtime information such as cache misses. Scheduling is, therefore, inherently conservative.
 - ▶ Branch and memory prediction is more difficult.
 - ▶ VLIW performance is highly dependent on the compiler. A number of techniques such as loop unrolling, speculative execution, branch prediction are critical.
 - ▶ Typical VLIW processors are limited to 4-way to 8-way parallelism.
- 

Limitations of Memory System Performance

- Memory system, and not processor speed, is often the bottleneck for many applications.
- Memory system performance is largely captured by two parameters, latency and bandwidth.
- Latency is the time from the issue of a memory request to the time the data is available at the processor.
- Bandwidth is the rate at which data can be pumped to the processor by the memory system.

Memory System Performance: Bandwidth and Latency

- It is very important to understand the difference between latency and bandwidth.
- Consider the example of a fire-hose. If the water comes out of the hose two seconds after the hydrant is turned on, the latency of the system is two seconds.
- Once the water starts flowing, if the hydrant delivers water at the rate of 5 gallons/second, the bandwidth of the system is 5 gallons/second.
- If you want immediate response from the hydrant, it is important to reduce latency.
- If you want to fight big fires, you want high bandwidth.

Memory Latency: An Example

Consider a processor operating at 1 GHz (1 ns clock) connected to a DRAM with a latency of 100 ns (no caches). Assume that the processor has two multiply-add units and is capable of executing four instructions in each cycle of 1 ns. The following observations follow:

- The peak processor rating is 4 GFLOPS.
- Since the memory latency is equal to 100 cycles and block size is one word, every time a memory request is made, the processor must wait 100 cycles before it can process the data.

Memory Latency: An Example

On the above architecture, consider the problem of computing a dot-product of two vectors.

- A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
- It follows that the peak speed of this computation is limited to one floating point operation every 100 ns, or a speed of 10 MFLOPS, a very small fraction of the peak processor rating!

Improving Effective Memory Latency Using Caches

- Caches are small and fast memory elements between the processor and DRAM.
- This memory acts as a low-latency high-bandwidth storage.
- If a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
- The fraction of data references satisfied by the cache is called the cache *hit ratio* of the computation on the system.
- Cache hit ratio achieved by a code on a memory system often determines its performance.

Impact of Caches: Example

Consider the architecture from the previous example. In this case, we introduce a cache of size 32 KB with a latency of 1 ns or one cycle. We use this setup to multiply two matrices A and B of dimensions 32×32 . We have carefully chosen these numbers so that the cache is large enough to store matrices A and B , as well as the result matrix C .

Impact of Caches: Example (continued)

The following observations can be made about the problem:

- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately $200\ \mu\text{s}$.
- Multiplying two $n \times n$ matrices takes $2n^3$ operations. For our problem, this corresponds to 64K operations, which can be performed in 16K cycles (or $16\ \mu\text{s}$) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., $200 + 16\ \mu\text{s}$.
- This corresponds to a peak computation rate of $64\text{K}/216$ or 303 MFLOPS.

Impact of Caches

- Repeated references to the same data item correspond to temporal locality.
- In our example, we had $O(n^2)$ data accesses and $O(n^3)$ computation. This asymptotic difference makes the above example particularly desirable for caches.
- Data reuse is critical for cache performance.

Impact of Memory Bandwidth

- Memory bandwidth is determined by the bandwidth of the memory bus as well as the memory units.
- Memory bandwidth can be improved by increasing the size of memory blocks.
- The underlying system takes l time units (where l is the latency of the system) to deliver b units of data (where b is the block size).

Impact of Memory Bandwidth: Example

Consider the same setup as before, except in this case, the block size is 4 words instead of 1 word. We repeat the dot-product computation in this scenario:

- Assuming that the vectors are laid out linearly in memory, eight FLOPs (four multiply-adds) can be performed in 200 cycles.
- This is because a single memory access fetches four consecutive words in the vector.
- Therefore, two accesses can fetch four elements of each of the vectors. This corresponds to a FLOP every 25 ns, for a peak speed of 40 MFLOPS.

Impact of Memory Bandwidth

- It is important to note that increasing block size does not change latency of the system.
- Physically, the scenario illustrated here can be viewed as a wide data bus (4 words or 128 bits) connected to multiple memory banks.
- In practice, such wide buses are expensive to construct.
- In a more practical system, consecutive words are sent on the memory bus on subsequent bus cycles after the first word is retrieved.

Impact of Memory Bandwidth

- The above examples clearly illustrate how increased bandwidth results in higher peak computation rates.
- The data layouts were assumed to be such that consecutive data words in memory were used by successive instructions (spatial locality of reference).
- If we take a data-layout centric view, computations must be reordered to enhance spatial locality of reference.

Impact of Memory Bandwidth: Example

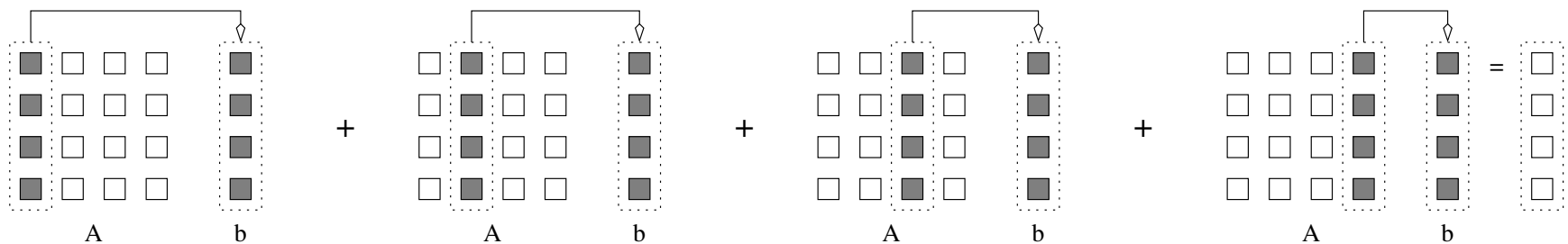
Consider the following code fragment:

```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
    for (j = 0; j < 1000; j++)  
        column_sum[i] += b[j][i];
```

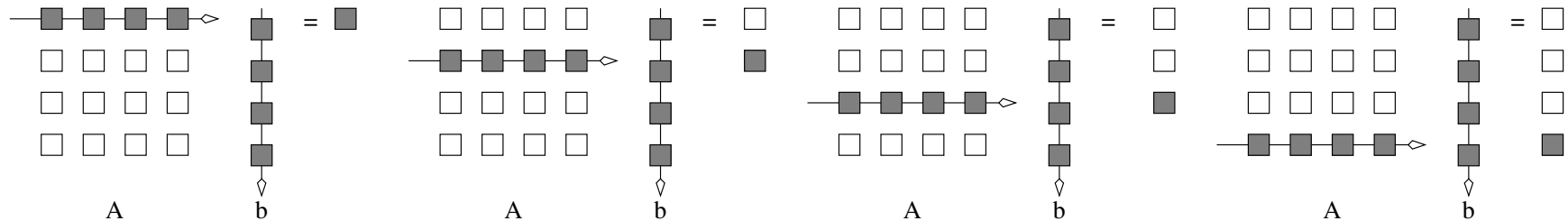
The code fragment sums columns of the matrix `b` into a vector `column_sum`.

Impact of Memory Bandwidth: Example

- The vector `column_sum` is small and easily fits into the cache
- The matrix `b` is accessed in a column order.
- The strided access results in very poor performance.



(a) Column major data access



(b) Row major data access.

Multiplying a matrix with a vector: (a) multiplying column-by-column, keeping a running sum; (b) computing each element of the result as a dot product of a row of the matrix with the vector.

Impact of Memory Bandwidth: Example

We can fix the above code as follows:

```
for (i = 0; i < 1000; i++)  
    column_sum[i] = 0.0;  
for (j = 0; j < 1000; j++)  
    for (i = 0; i < 1000; i++)  
        column_sum[i] += b[j][i];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better.

Memory System Performance: Summary

The series of examples presented in this section illustrate the following concepts:

- Exploiting spatial and temporal locality in applications is critical for amortizing memory latency and increasing effective memory bandwidth.
- The ratio of the number of operations to number of memory accesses is a good indicator of anticipated tolerance to memory bandwidth.
- Memory layouts and organizing computation appropriately can make a significant impact on the spatial and temporal locality.

Alternate Approaches for Hiding Memory Latency

Consider the problem of browsing the web on a very slow network connection. We deal with the problem in one of three possible ways:

- we anticipate which pages we are going to browse ahead of time and issue requests for them in advance;
- we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others; or
- we access a whole bunch of pages in one go – amortizing the latency across various accesses.

The first approach is called *prefetching*, the second *multithreading*, and the third one corresponds to spatial locality in accessing memory words.

Multithreading for Latency Hiding

A thread is a single stream of control in the flow of a program. We illustrate threads with a simple example:

```
for (i = 0; i < n; i++)  
    c[i] = dot_product(get_row(a, i), b);
```

Each dot-product is independent of the other, and therefore represents a concurrent unit of execution. We can safely rewrite the above code segment as:

```
for (i = 0; i < n; i++)  
    c[i] = create_thread(dot_product, get_row(a, i), b);
```

Multithreading for Latency Hiding: Example

- In the code, the first instance of this function accesses a pair of vector elements and waits for them.
- In the meantime, the second instance of this function can access two other vector elements in the next cycle, and so on.
- After l units of time, where l is the latency of the memory system, the first function instance gets the requested data from memory and can perform the required computation.
- In the next cycle, the data items for the next function instance arrive, and so on. In this way, in every clock cycle, we can perform a computation.

Multithreading for Latency Hiding

- The execution schedule in the previous example is predicated upon two assumptions: the memory system is capable of servicing multiple outstanding requests, and the processor is capable of switching threads at every cycle.
- It also requires the program to have an explicit specification of concurrency in the form of threads.
- Machines such as the HEP and Tera rely on multithreaded processors that can switch the context of execution in every cycle. Consequently, they are able to hide latency effectively.

Prefetching for Latency Hiding

- Misses on loads cause programs to stall.
- Why not advance the loads so that by the time the data is actually needed, it is already there!
- The only drawback is that you might need more space to store advanced loads.
- However, if the advanced loads are overwritten, we are no worse than before!

Explicitly Parallel Platforms

Dichotomy of Parallel Computing Platforms

- An explicitly parallel program must specify concurrency and interaction between concurrent subtasks.
- The former is sometimes also referred to as the control structure and the latter as the communication model.

Control Structure of Parallel Programs

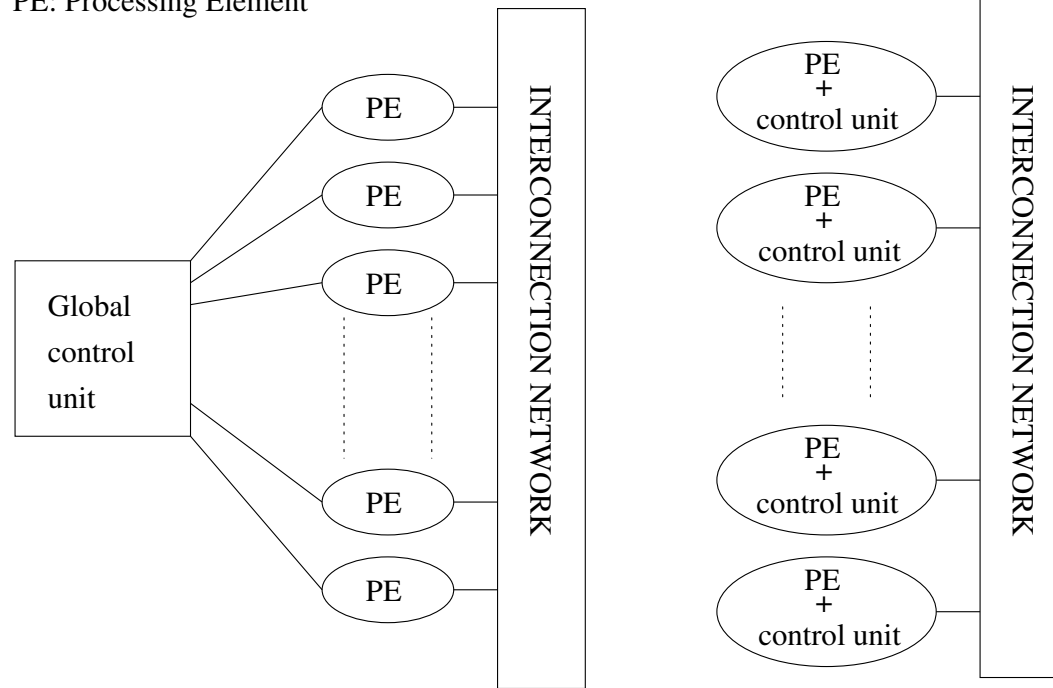
- Parallelism can be expressed at various levels of granularity – from instruction level to processes.
- Between these extremes exist a range of models, along with corresponding architectural support.

Control Structure of Parallel Programs

- Processing units in parallel computers either operate under the centralized control of a single control unit or work independently.
- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as single instruction stream, multiple data stream (SIMD).
- If each processor has its own control control unit, each processor can execute different instructions on different data items. This model is called multiple instruction stream, multiple data stream (MIMD).

SIMD and MIMD Processors

PE: Processing Element



(a)

(b)

A typical SIMD architecture (a) and a typical MIMD architecture (b).

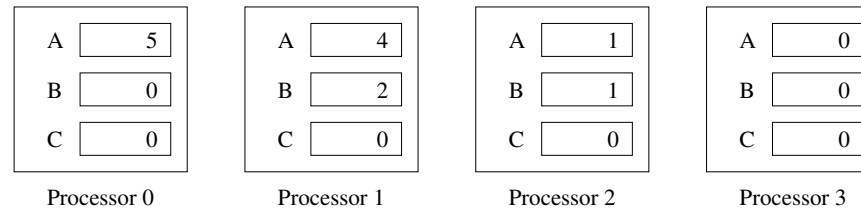
SIMD Processors

- Some of the earliest parallel computers such as the Illiac IV, MPP, DAP, CM-2, and MasPar MP-1 belonged to this class of machines.
- Variants of this concept have found use in co-processing units such as the MMX units in Intel processors and DSP chips such as the Sharc.
- SIMD relies on the regular structure of computations (such as those in image processing).
- It is often necessary to selectively turn off operations on certain data items. For this reason, most SIMD programming paradigms allow for an “activity mask”, which determines if a processor should participate in a computation or not.

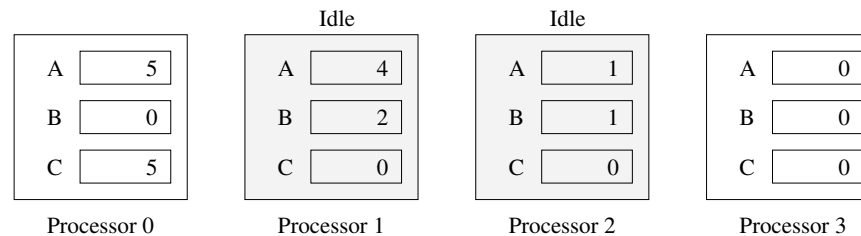
Conditional Execution in SIMD Processors

```
if (B == 0)
    C = A;
else
    C = A/B;
```

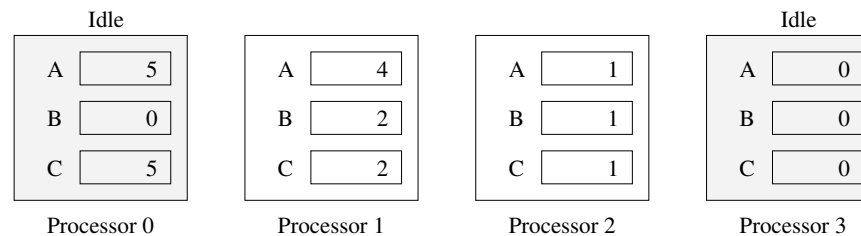
(a)



Initial values



Step 1



Step 2

(b)

Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.

MIMD Processors

- In contrast to SIMD processors, MIMD processors can execute different programs on different processors.
- A variant of this, called single program multiple data streams (SPMD) executes the same program on different processors.
- It is easy to see that SPMD and MIMD are closely related in terms of programming flexibility and underlying architectural support.
- Examples of such platforms include current generation Sun Ultra Servers, SGI Origin Servers, multiprocessor PCs, workstation clusters, and the IBM SP.

SIMD-MIMD Comparison

- SIMD computers require less hardware than MIMD computers (single control unit).
- However, since SIMD processors are specially designed, they tend to be expensive and have long design cycles.
- Not all applications are naturally suited to SIMD processors.
- In contrast, platforms supporting the SPMD paradigm can be built from inexpensive off-the-shelf components with relatively little effort in a short amount of time.

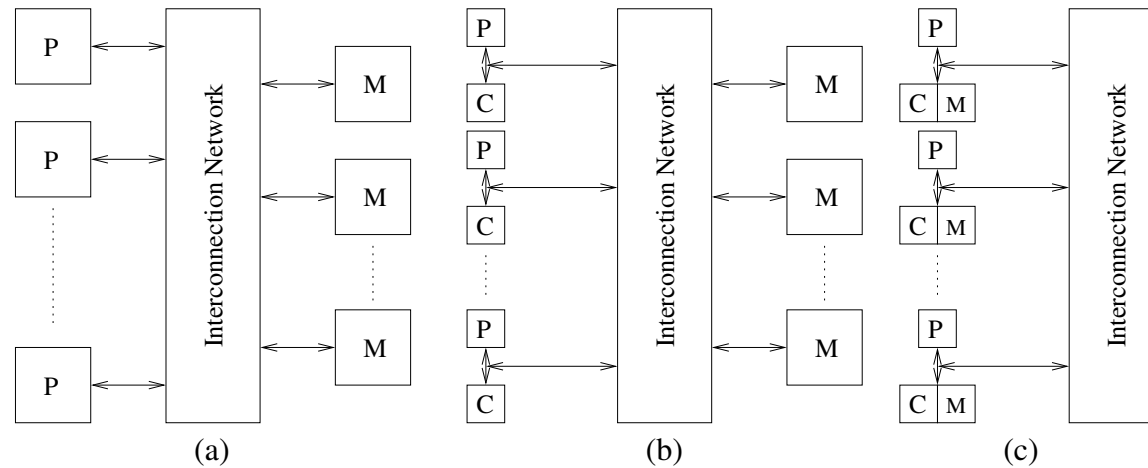
Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks – accessing a shared data space and exchanging messages.
- Platforms that provide a shared data space are called shared-address-space machines or multiprocessors.
- Platforms that support messaging are also called message passing platforms or multicomputers.

Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system global or local is identical, the platform is classified as a uniform memory access (UMA), else, a non-uniform memory access (NUMA) machine.

NUMA and UMA Shared-Address-Space Platforms



Typical shared-address-space architectures: (a) Uniform-memory-access shared-address-space computer; (b) Uniform-memory-access shared-address-space computer with caches and memories; (c) Non-uniform-memory-access shared-address-space computer with local memory only.

NUMA and UMA Shared-Address-Space Platforms

- The distinction between NUMA and UMA platforms is important from the point of view of algorithm design. NUMA machines require locality from underlying algorithms for performance.
- Programming these platforms is easier since reads and writes are implicitly visible to other processors.
- However, read-write data to shared data must be coordinated (this will be discussed in greater detail when we talk about threads programming).
- Caches in such machines require coordinated access to multiple copies. This leads to the cache coherence problem.
- A weaker model of these machines provides an address map, but not coordinated access. These models are called non cache coherent shared address space machines.

Shared-Address-Space vs. Shared Memory Machines

- It is important to note the difference between the terms shared address space and shared memory.
- We refer to the former as a programming abstraction and to the latter as a physical machine attribute.
- It is possible to provide a shared address space using a physically distributed memory.

Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.
- These platforms are programmed using (variants of) send and receive primitives.
- Libraries such as MPI and PVM provide such primitives.

Message Passing vs. Shared Address Space Platforms

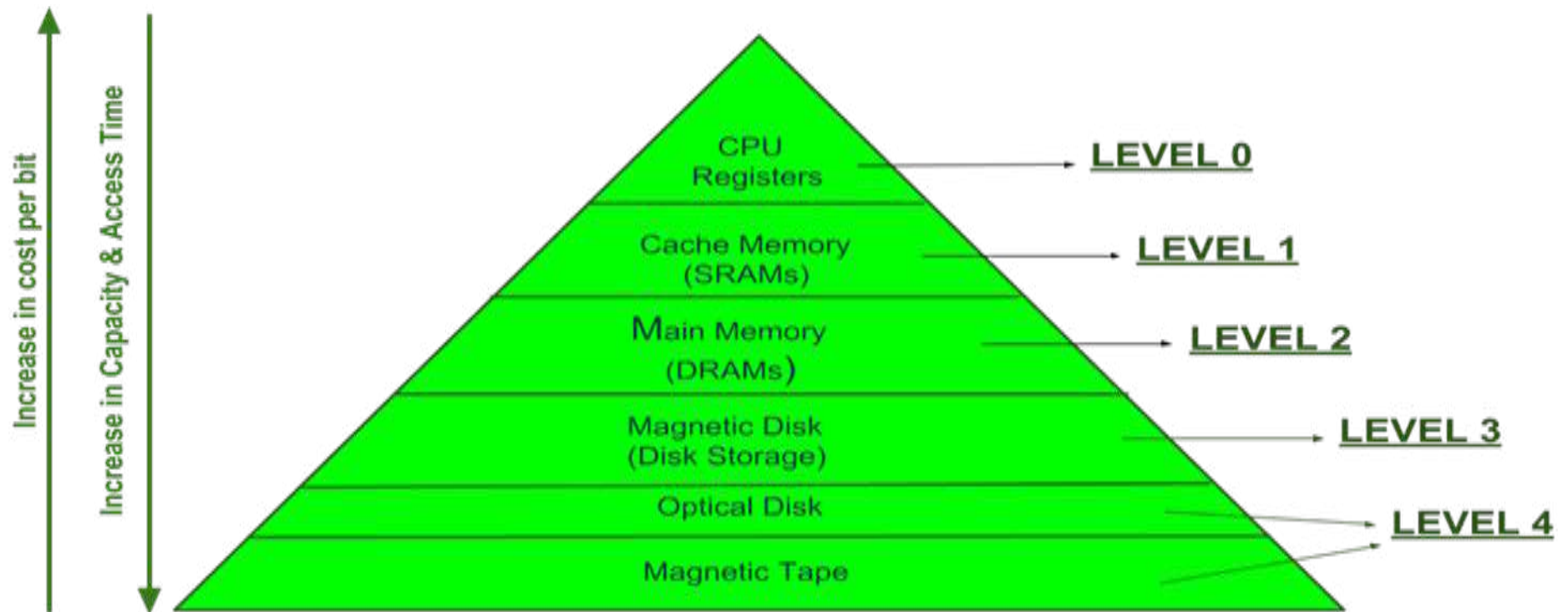
- Message passing requires little hardware support, other than a network.
- Shared address space platforms can easily emulate message passing. The reverse is more difficult to do (in an efficient manner).

Physical Organization of Parallel Platforms

We begin this discussion with an ideal parallel machine called Parallel Random Access Machine, or PRAM.

Physical Organization of Parallel Platforms

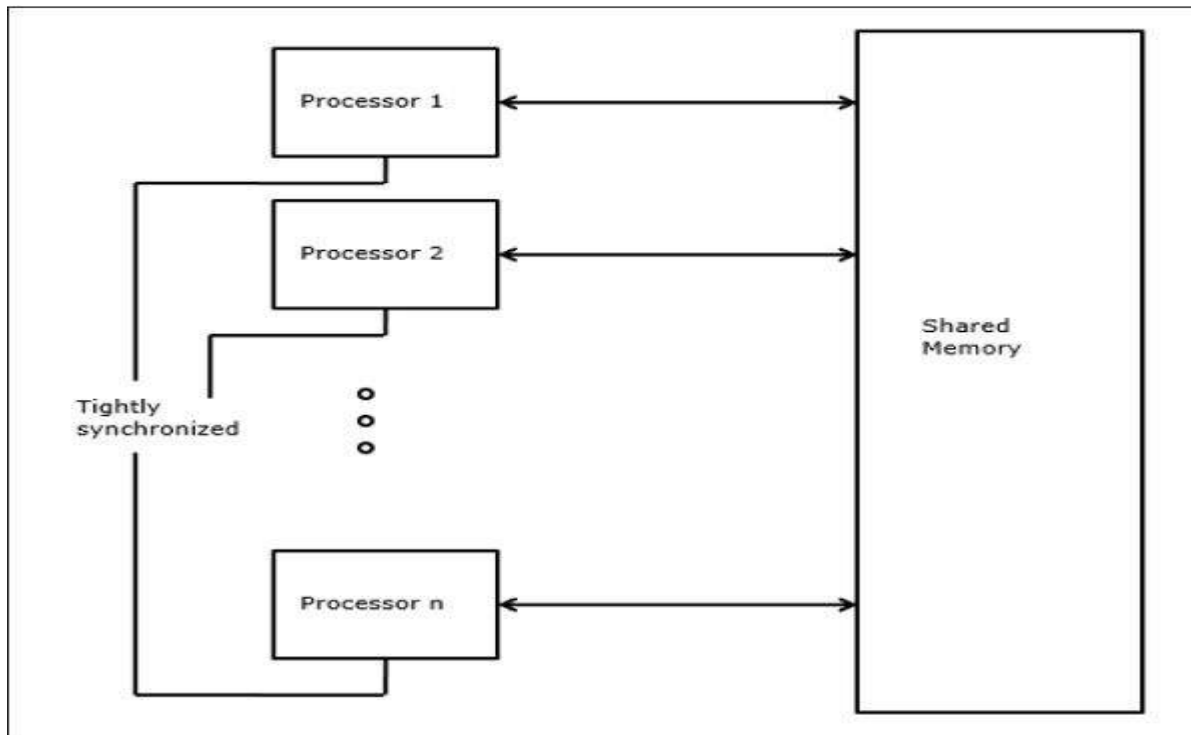
We begin this discussion with an ideal parallel machine called Parallel Random Access Machine, or PRAM.



MEMORY HIERARCHY DESIGN

Architecture of an Ideal Parallel Computer

- A natural extension of the Random Access Machine (RAM) serial architecture is the Parallel Random Access Machine, or PRAM.
- PRAMs consist of p processors and a global memory of unbounded size that is uniformly accessible to all processors.
- Processors share a common clock but may execute different instructions in each cycle.



Architecture of an Ideal Parallel Computer

Depending on how simultaneous memory accesses are handled, PRAMs can be divided into four subclasses.

- Exclusive-read, exclusive-write (EREW) PRAM.
- Concurrent-read, exclusive-write (CREW) PRAM.
- Exclusive-read, concurrent-write (ERCW) PRAM.
- Concurrent-read, concurrent-write (CRCW) PRAM.

Architecture of an Ideal Parallel Computer

What does concurrent write mean, anyway?

- Common: write only if all values are identical.
- Arbitrary: write the data from a randomly selected processor.
- Priority: follow a predetermined priority order.
- Sum: Write the sum of all data items.

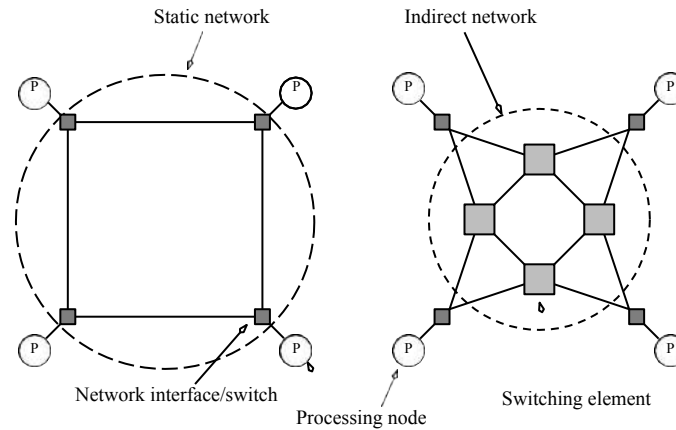
Physical Complexity of an Ideal Parallel Computer

- Processors and memories are connected via switches.
- Since these switches must operate in $O(1)$ time at the level of words, for a system of p processors and m words, the switch complexity is $O(mp)$.
- Clearly, for meaningful values of p and m , a true PRAM is not realizable.

Interconnection Networks for Parallel Computers

- Interconnection networks carry data between processors and to memory.
- Interconnects are made of switches and links (wires, fiber).
- Interconnects are classified as static or dynamic.
- Static networks consist of point-to-point communication links among processing nodes and are also referred to as *direct* networks.
- Dynamic networks are built using switches and communication links. Dynamic networks are also referred to as *indirect* networks.

Static and Dynamic Interconnection Networks



Classification of interconnection networks: (a) a static network; and (b) a dynamic network.

Interconnection Networks

- Switches map a fixed number of inputs to outputs.
- The total number of ports on a switch is the *degree* of the switch.
- The cost of a switch grows as the square of the degree of the switch, the peripheral hardware linearly as the degree, and the packaging costs linearly as the number of pins.

Interconnection Networks: Network Interfaces

- Processors talk to the network via a network interface.
- The network interface may hang off the I/O bus or the memory bus.
- In a physical sense, this distinguishes a cluster from a tightly coupled multicomputer.
- The relative speeds of the I/O and memory buses impact the performance of the network.

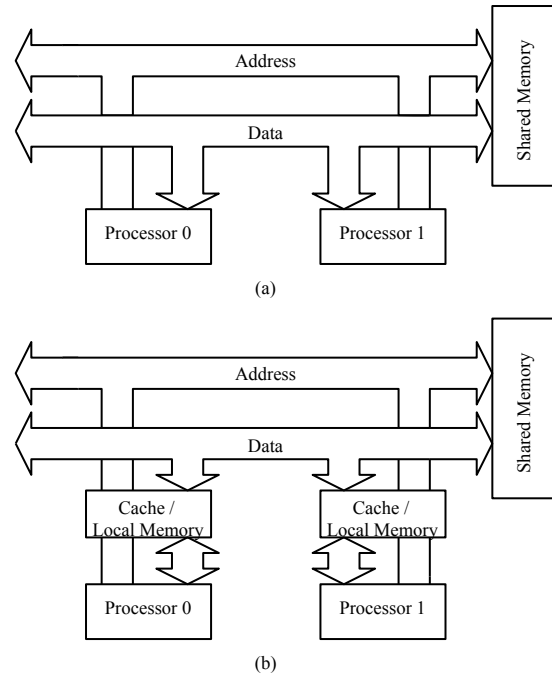
Network Topologies

- A variety of network topologies have been proposed and implemented.
- These topologies tradeoff performance for cost.
- Commercial machines often implement hybrids of multiple topologies for reasons of packaging, cost, and available components.

Network Topologies: Buses

- Some of the simplest and earliest parallel machines used buses.
- All processors access a common bus for exchanging data.
- The distance between any two nodes is $O(1)$ in a bus. The bus also provides a convenient broadcast media.
- However, the bandwidth of the shared bus is a major bottleneck.
- Typical bus based machines are limited to dozens of nodes. Sun Enterprise servers and Intel Pentium based shared-bus multiprocessors are examples of such architectures.

Network Topologies: Buses

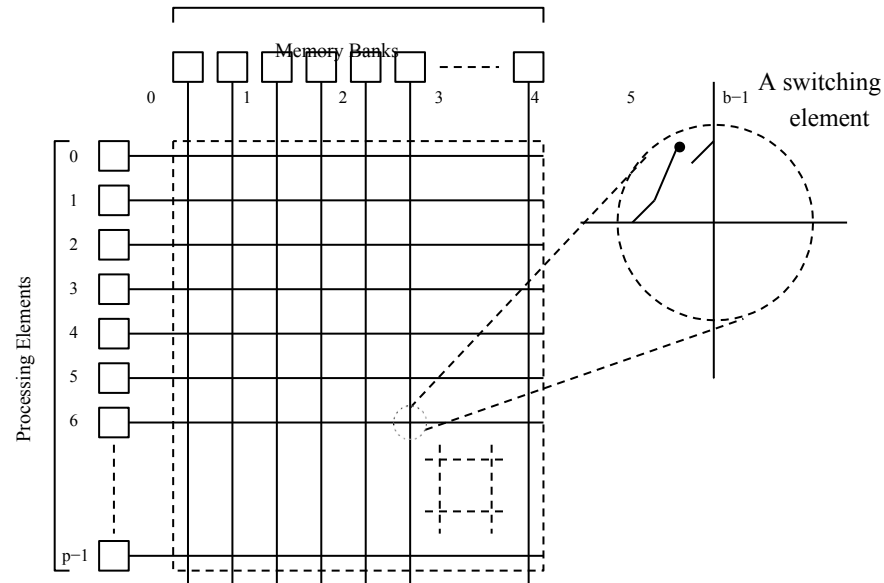


Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

Since much of the data accessed by processors is local to the processor, a local memory can improve the performance of bus-based machines.

Network Topologies: Crossbars

A crossbar network uses an $p \times m$ grid of switches to connect p inputs to m outputs in a non-blocking manner.



A completely non-blocking crossbar network connecting
 p
processors to b memory banks.

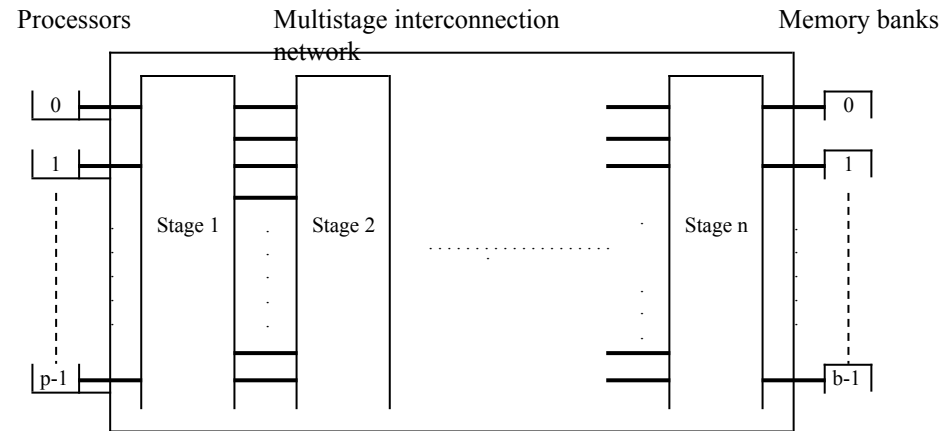
Network Topologies: Crossbars

- The cost of a crossbar of p processors grows as $O(p^2)$.
- This is generally difficult to scale for large values of p .
- Examples of machines that employ crossbars include the Sun Ultra HPC 10000 and the Fujitsu VPP500.

Network Topologies: Multistage Networks

- Crossbars have excellent performance scalability but poor cost scalability.
- Buses have excellent cost scalability, but poor performance scalability.
- Multistage interconnects strike a compromise between these extremes.

Network Topologies: Multistage Networks



The schematic of a typical multistage interconnection network.

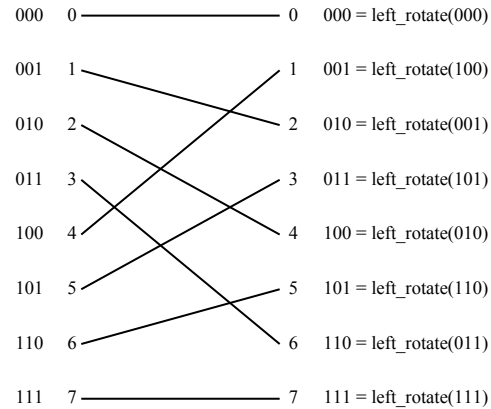
Network Topologies: Multistage Omega Network

- One of the most commonly used multistage interconnects is the Omega network.
- This network consists of $\log p$ stages, where p is the number of inputs/outputs.
- At each stage, input i is connected to output j if:

$$j = \begin{cases} 2i, & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p, & p/2 \leq i \leq p - 1 \end{cases}$$

Network Topologies: Multistage Omega Network

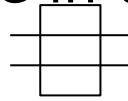
Each stage of the Omega network implements a perfect shuffle as follows:



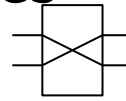
A perfect shuffle interconnection for eight inputs and outputs.

Network Topologies: Multistage Omega Network

- The perfect shuffle patterns are connected using 2×2 switches.
- The switches operate in two modes – crossover or passthrough.



(a)



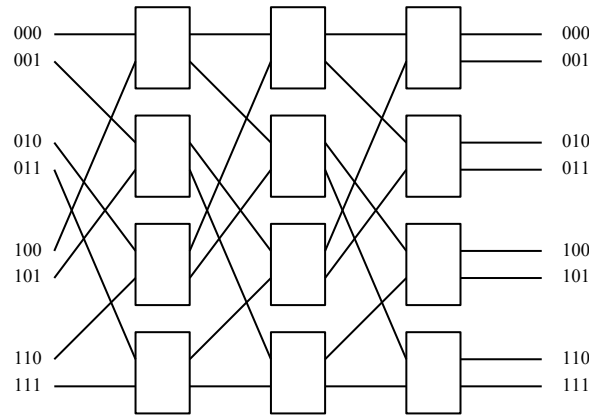
(b)

Two switching configurations of the 2×2 switch: (a) Pass-through;

(b) Cross-over.

Network Topologies: Multistage Omega Network

A complete Omega network with the perfect shuffle interconnects and switches can now be illustrated:



A complete omega network connecting eight inputs and eight outputs.

An omega network has $p/2 \times \log p$ switching nodes, and the cost of such a network grows as $\Theta(p \log p)$.

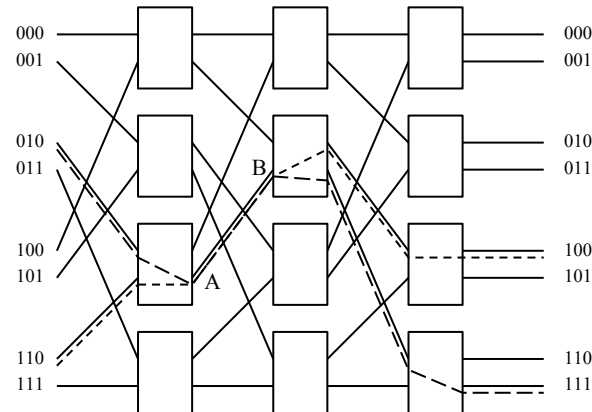
Network Topologies: Multistage Omega Network

– Routing

- Let s be the binary representation of the source and d be that of the destination processor.
- The data traverses the link to the first switching node. If the most significant bits of s and t are the same, then the data is routed in pass-through mode by the switch else, it switches to crossover.
- This process is repeated for each of the $\log p$ switching stages.
- Note that this is not a non-blocking switch.

Network Topologies: Multistage Omega Network

– Routing



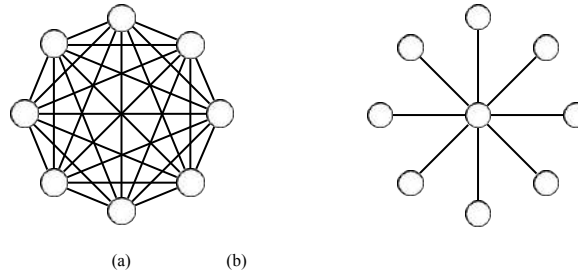
An example of blocking in omega network: one of the messages (010 to 111 or 110 to 100) is blocked at link AB.

Network Topologies: Completely Connected Network

- Each processor is connected to every other processor.
- The number of links in the network scales as $O(p^2)$.
- While the performance scales very well, the hardware complexity is not realizable for large values of p .
- In this sense, these networks are static counterparts of crossbars.

Network Topologies: Completely Connected and Star Connected Networks

Example of an 8-node completely connected network.



(a) A completely-connected network of eight nodes; (b) a Star connected network of nine nodes.

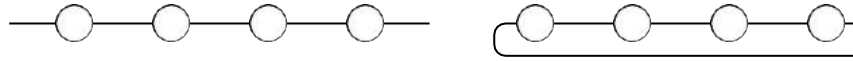
Network Topologies: Star Connected Network

- Every node is connected only to a common node at the center.
- Distance between any pair of nodes is $O(1)$. However, the central node becomes a bottleneck.
- In this sense, star connected networks are static counterparts of buses.

Network Topologies: Linear Arrays, Meshes, and k - d Meshes

- In a linear array, each node has two neighbors, one to its left and one to its right. If the nodes at either end are connected, we refer to it as a 1-D torus or a ring.
- A generalization to 2 dimensions has nodes with 4 neighbors, to the north, south, east, and west.
- A further generalization to d dimensions has nodes with $2d$ neighbors.
- A special case of a d -dimensional mesh is a hypercube. Here,
 $d = \log p$, where p is the total number of nodes.

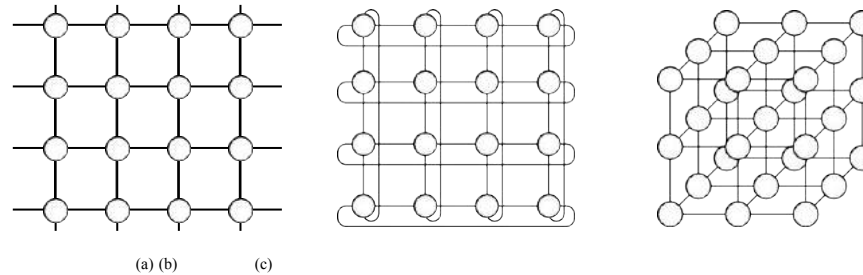
Network Topologies: Linear Arrays



(a) (b)

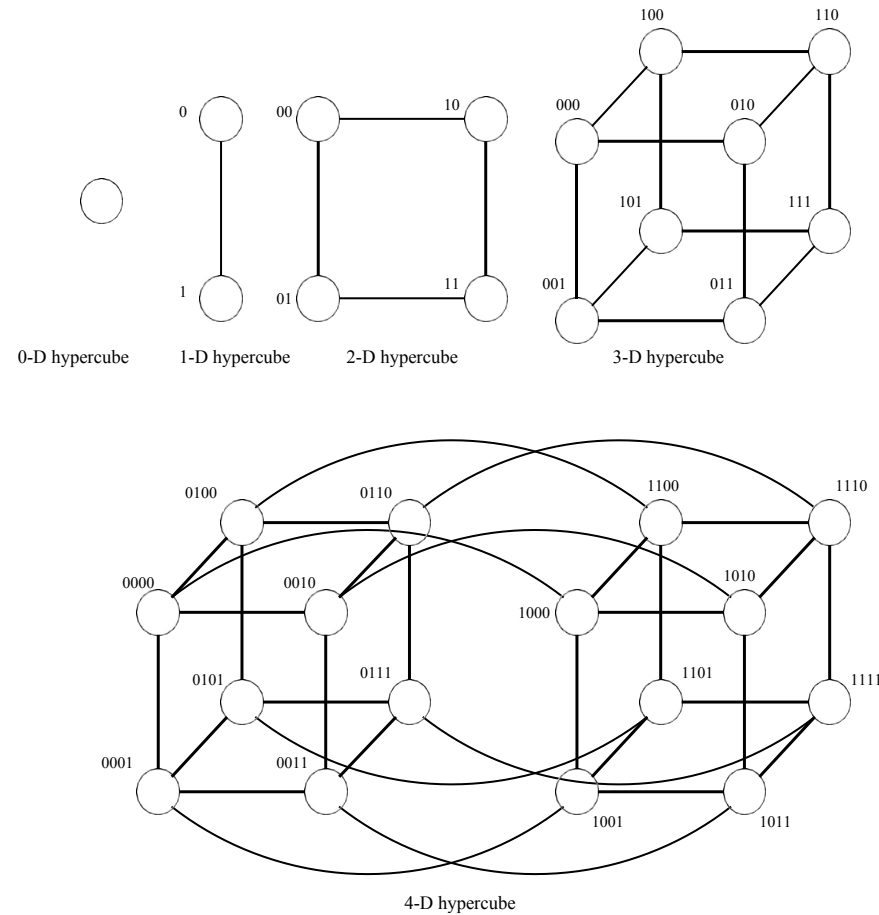
Linear arrays: (a) with no wraparound links; (b) with wraparound link.

Network Topologies: Two- and Three Dimensional Meshes



Two and three dimensional meshes: (a) 2-D mesh with no wraparound; (b) 2-D mesh with wraparound link (2-D torus); and
(c) a 3-D mesh with no wraparound.

Network Topologies: Hypercubes and their Construction

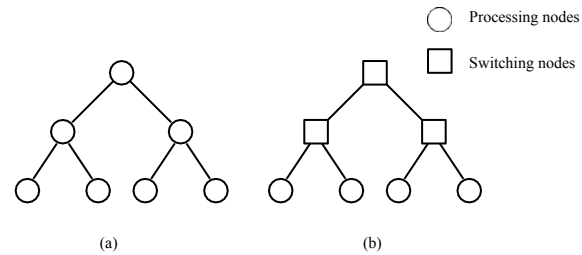


Construction of hypercubes from hypercubes of lower dimension.

Network Topologies: Properties of Hypercubes

- The distance between any two nodes is at most $\log p$.
- Each node has $\log p$ neighbors.
- The distance between two nodes is given by the number of bit positions at which the two nodes differ.

Network Topologies: Tree-Based Networks

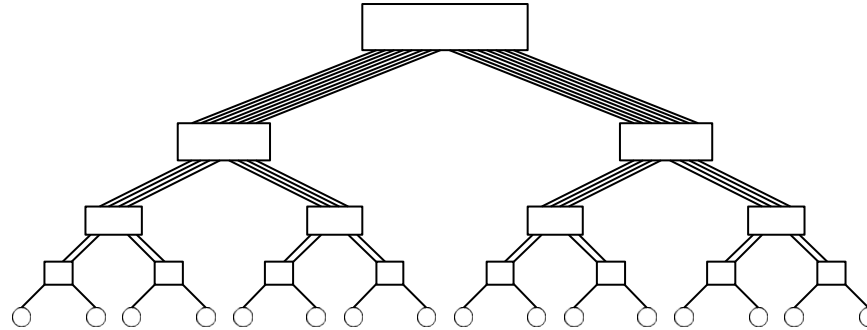


Complete binary tree networks: (a) a static tree network;
and
(b) a dynamic tree network.

Network Topologies: Tree Properties

- The distance between any two nodes is no more than $2 \log p$.
- Links higher up the tree potentially carry more traffic than those at the lower levels.
- For this reason, a variant called a fat-tree, fattens the links as we go up the tree.
- Trees can be laid out in 2D with no wire crossings. This is an interesting property of trees.

Network Topologies: Fat Trees



A fat tree network of 16 processing nodes.

Evaluating Static Interconnection Networks

- Diameter: The distance between the farthest two nodes in the network. The diameter of a linear array is $p - 1$, that of a mesh is $2(\sqrt{p} - 1)$, that of a tree and hypercube is $\log p$, and that of a completely connected network is $O(1)$.
- Bisection Width: The minimum number of wires you must cut to divide the network into two equal parts. The bisection width of a linear array and tree is 1, that of a mesh is \sqrt{p} , that of a hypercube is $p/2$ and that of a completely connected network is $p^2/4$.
- Cost: The number of links or switches (whichever is asymptotically higher) is a meaningful measure of the cost. However, a number of other factors, such as the ability to lay out the network, the length of wires, etc., also factor in to the cost.

Evaluating Static Interconnection Networks

Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p - 1$	$p(p - 1)/2$
Star	2	1	1	$p - 1$
Complete binary tree	$2 \log((p + 1)/2)$	1	1	$p - 1$
Linear array	$p - 1$	1	1	$p - 1$
2-D mesh, no wraparound	$2(\sqrt{p} - 1)$	\sqrt{p}	2	$2(p - \sqrt{p})$
2-D wraparound mesh	$2\lceil \sqrt{p/2} \rceil$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p \log p)/2$
Wraparound k -ary d -cube	$d\lceil k/2 \rceil$	$2k^{d-1}$	$2d$	dp

Evaluating Dynamic Interconnection Networks

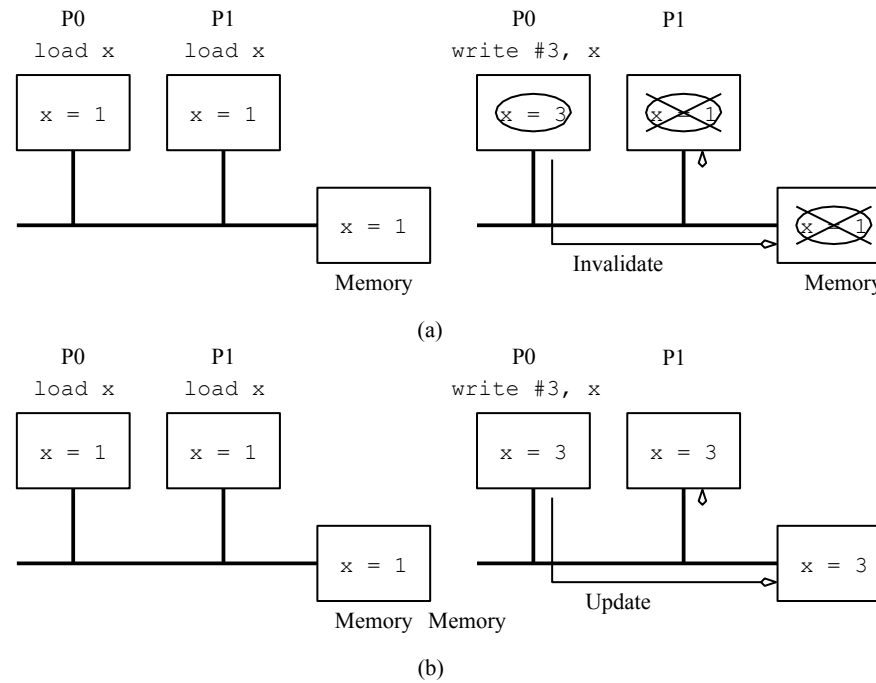
1pt Network	Diameter	Bisection Width	Arc Connectivity	Cost (No. of links)
Crossbar	1	p	1	p^2
Omega Network	$\log p$	$p/2$	2	$p/2$
Dynamic Tree	$2 \log p$	1	2	$p - 1$

Cache Coherence in Multiprocessor Systems

- Interconnects provide basic mechanisms for data transfer.
- In the case of shared address space machines, additional hardware is required to coordinate access to data that might have multiple copies in the network.
- The underlying technique must provide some guarantees on the semantics.
- This guarantee is generally one of serializability, i.e., there exists some serial order of instruction execution that corresponds to the parallel schedule.

Cache Coherence in Multiprocessor Systems

When the value of a variable is changes, all its copies must either be invalidated or updated.



Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

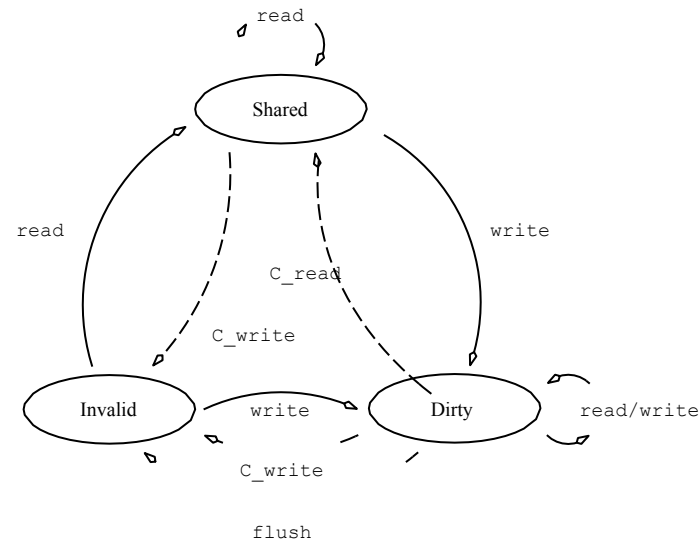
Cache Coherence: Update and Invalidate Protocols

- If a processor just reads a value once and does not need it again, an update protocol may generate significant overhead.
- If two processors make interleaved test and updates to a variable, an update protocol is better.
- Both protocols suffer from false sharing overheads (two words that are not shared, however, they lie on the same cache line).
- Most current machines use invalidate protocols.

Maintaining Coherence Using Invalidate Protocols

- Each copy of a data item is associated with a state.
- One example of such a set of states is, shared, invalid, or dirty.
- In shared state, there are multiple valid copies of the data item (and therefore, an invalidate would have to be generated on an update).
- In dirty state, only one copy exists and therefore, no invalidates need to be generated.
- In invalid state, the data copy is invalid, therefore, a read generates a data request (and associated state changes).

Maintaining Coherence Using Invalidate Protocols



State diagram of a simple three-state coherence protocol.

Maintaining Coherence Using Invalidate Protocols

Time
↓

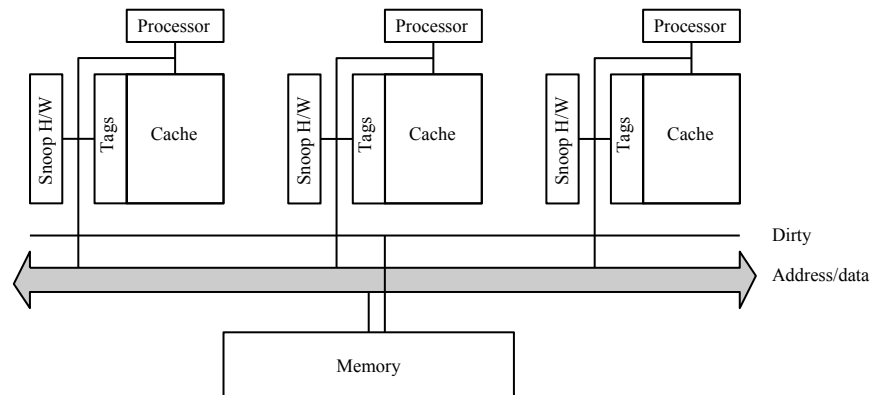
Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x	read y	x = 5, S	y = 12, S	x = 5, S y = 12, S
x = x + 1	y = y + 1	x = 6, D	y = 13, D	x = 5, I y = 12, I
read y	read x	y = 13, S x = 6, S	y = 13, S x = 6, S	y = 13, S x = 6, S
x = x + y	y = x + y	x = 19, D y = 13, I	x = 6, I y = 19, D	x = 6, I y = 13, I
x = x + 1	y = y + 1	x = 20, D y = 13, I	x = 6, I y = 20, D	x = 6, I y = 13, I

Example of parallel program execution with the simple three-state coherence protocol.

Snoopy Cache Systems

How are invalidates sent to the right processors?

In snoopy caches, there is a broadcast media that listens to all invalidates and read requests and performs appropriate coherence operations locally.



A simple snoopy bus based cache coherence system.

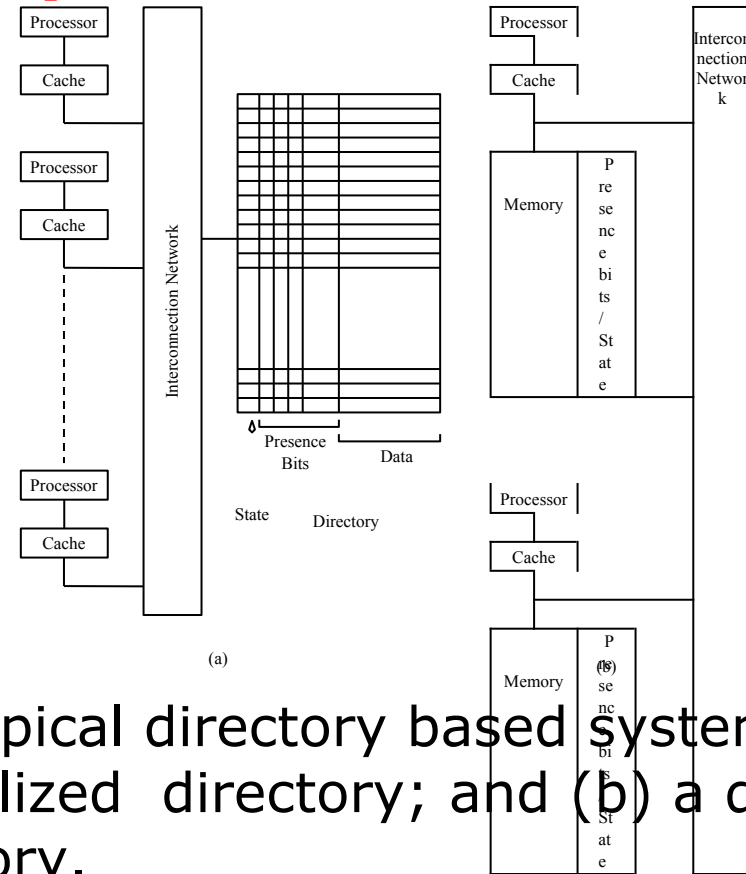
Performance of Snoopy Caches

- Once copies of data are tagged dirty, all subsequent operations can be performed locally on the cache without generating external traffic.
- If a data item is read by a number of processors, it transitions to the shared state in the cache and all subsequent read operations become local.
- If processors read and update data at the same time, they generate coherence requests on the bus – which is ultimately bandwidth limited.

Directory Based Systems

- In snoopy caches, each coherence operation is sent to all processors. This is an inherent limitation.
- Why not send coherence requests to only those processors that need to be notified?
- This is done using a directory, which maintains a presence vector for each data item (cache line) along with its global state.

Directory Based Systems



Architecture of typical directory based systems: (a) a centralized directory; and (b) a distributed directory.

Performance of Directory Based Schemes

- The need for a broadcast media is replaced by the directory.
- The additional bits to store the directory may add significant overhead.
- The underlying network must be able to carry all the coherence requests.
- The directory is a point of contention, therefore, distributed directory schemes must be used.

Analytical Modeling of Parallel Systems

Dr. B Krishna Priya

Topic Overview

- Sources of Overhead in Parallel Programs
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Parallel Platforms: Models (SIMD, MIMD, SPMD) ,
Communication (Shared Address Space vs. Message Passing)

Analytical Modeling: Sequential Execution Time

- The execution time of a sequential algorithm
 - Asymptotic execution time as a function of input size
 - identical on any serial platform

Example: Matrix Multiplication

```
int n = A.length;                                <-- cost = c0, 1 time
for (int i = 0; i < n; i++) {                     <-- cost = c1, n times
    for (int j = 0; j < n; j++) {                 <-- cost = c2, n*n times
        sum = 0;                                <-- cost = c3, n*n times
        for k = 0; k < n; k++)                   <-- cost = c4, n*n*n times
            sum = sum + A[i][k]*B[k][j];         <-- cost = c5, n*n*n times
        C[i][j] = sum;                          <-- cost = c6, n*n times
    }
}
```

Total number of operations:

$$\begin{aligned} &= c_0 + c_1 * n + (c_2 + c_3 + c_6) * n * n + (c_4 + c_5) * n * n * n \\ &= O(n^3) \end{aligned}$$

- Big-O Notation

- $O(1)$
- $O(N)$
- $O(N^2)$
- $O(N \log N)$
- $O(N^3)$
- ...

Count the number of operations

Analytical Modeling - Basics

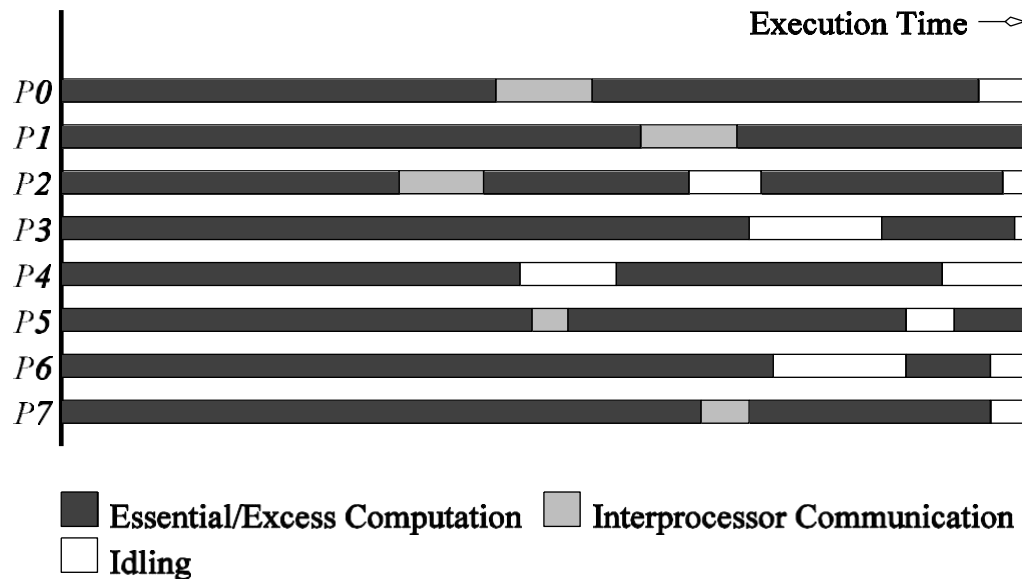
- A **sequential algorithm** is evaluated by its runtime (in general, **asymptotic runtime** as a function of input size).
- The asymptotic runtime of a sequential program is **identical on any serial platform**.
- The **parallel runtime** of a program depends on the **input size**, the **number of processors**, and the **communication parameters** of the machine.
- An algorithm must therefore be analyzed in the context of the **underlying platform**.
- A parallel system is a combination of a parallel algorithm and an underlying platform.

Analytical Modeling - Basics

- A number of performance measures are intuitive.
- **Wall clock time** - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. **But how does this scale when the number of processors is changed** of the program is ported to another machine altogether?
- **How much faster is the parallel version?** This begs the obvious followup question - whats the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look
- **Raw FLOPS (FLoating-point Operations Per Second)** - How good is FLOPS measure when it don't solve a problem?

Sources of Overhead in Parallel Programs

- If I use two processors, shouldnt my program run twice as fast?
- No - a number of overheads, including **wasted computation**, **communication**, **idling**, and **contention** cause degradation in performance.



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Sources of Overheads in Parallel Programs

- **Interprocess interactions:** Processors working on any non-trivial parallel problem will **need to talk to each other**.
- **Idling:** Processes may idle because of **load imbalance**, **synchronization**, or **serial components**.
- **Excess Computation:** This is computation **not performed by the serial version**. This might be because the **serial algorithm is difficult to parallelize**, or that some computations are repeated across processors **to minimize communication**.

Performance Metrics for Parallel Systems: Execution Time

- **Serial runtime** of a program is the **time elapsed between the beginning and the end** of its execution on a sequential computer.
- **The parallel runtime** is the time that elapses from the moment the **first processor starts** to the moment the **last processor finishes** execution.
- We denote the serial runtime by T_S and the parallel runtime by T_P .

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_s is the serial time.
- Observe that $T_{all} - T_s$ is then the total time spend by all processors combined in **non-useful work**. This is called the ***total overhead***.
- The total time collectively spent by all the processing elements
 $T_{all} = p T_P$ (p is the number of processors).
- The overhead function (T_o) is therefore given by

$$T_o = p T_P - T_s \quad (1)$$

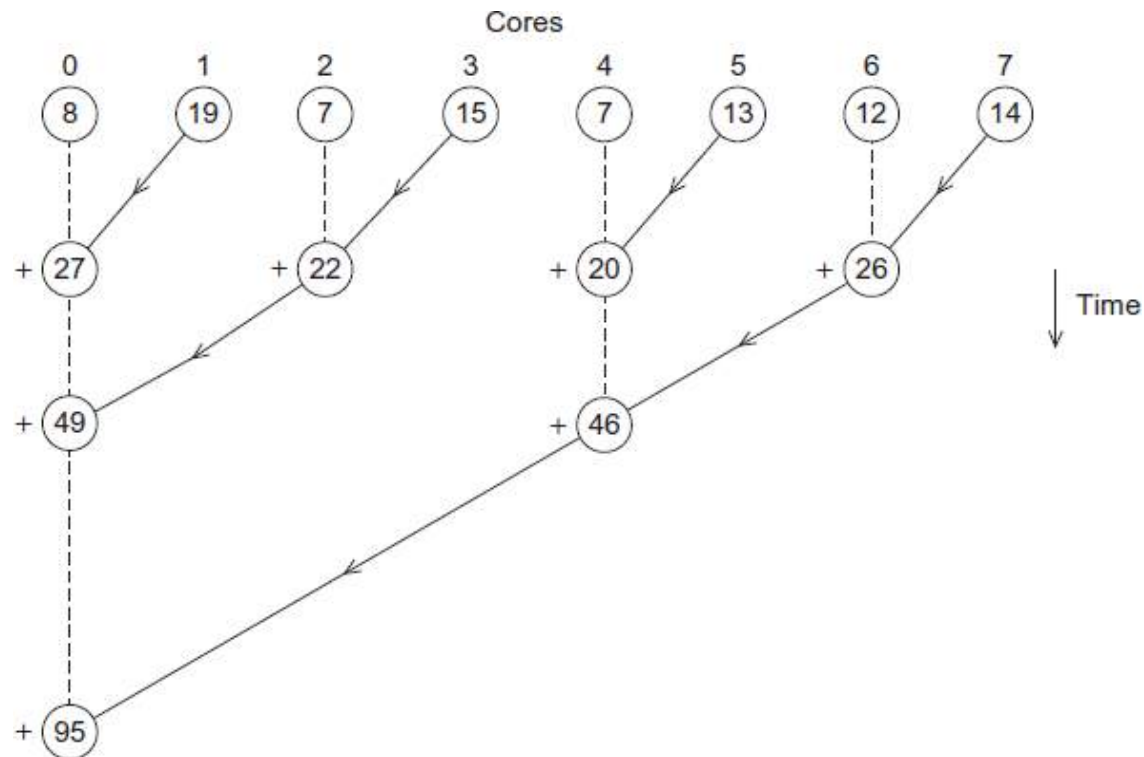
Performance Metrics for Parallel Systems: Speedup

- What is the benefit from parallelism?
- **Speedup (S)** is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

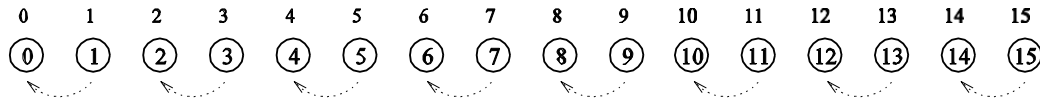
$$S = \frac{T_S}{T_P}$$

Performance Metrics: Example

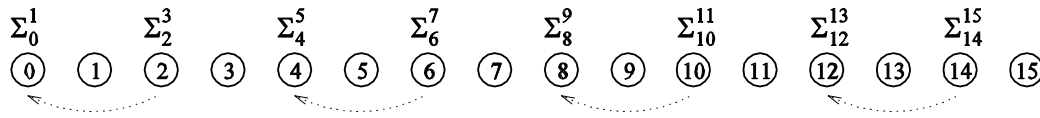
- Consider the problem of **adding n numbers** by using **n** processing elements.
- If **n** is a power of two, we can perform this operation in **$\log n$** steps by **propagating partial sums** up a logical binary tree of processors.



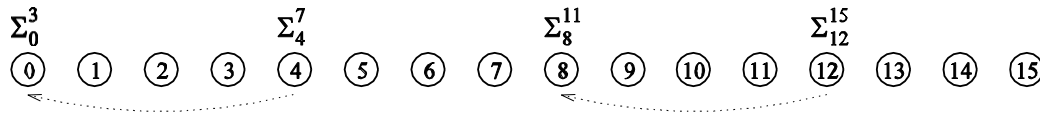
Performance Metrics: Example



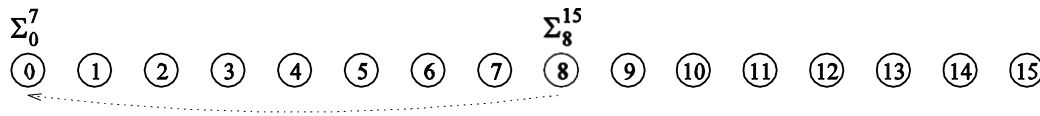
(a) Initial data distribution and the first communication step



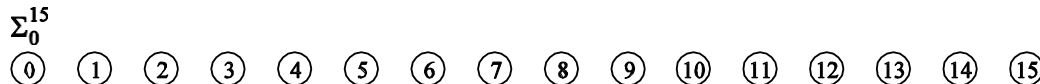
(b) Second communication step



(c) Third communication step



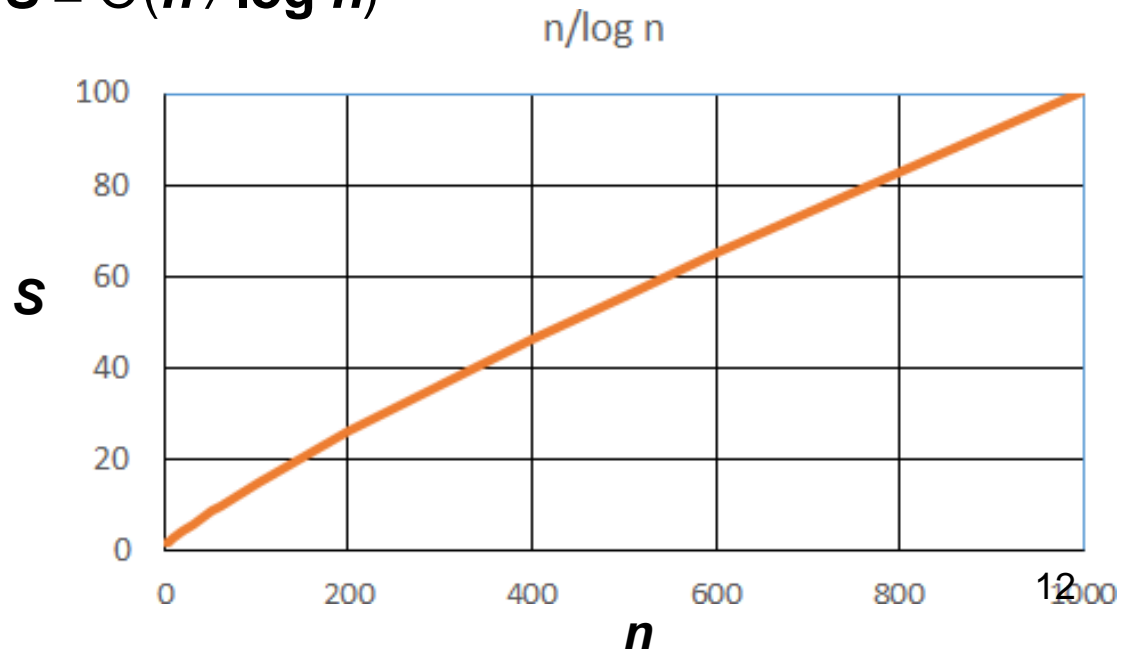
(d) Fourth communication step



(e) Accumulation of the sum at processing element 0 after the final communication

Performance Metrics: Example (continued)

- If an addition takes **constant time**, say, t_c and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_P = \Theta(\log n)$
- We know that $T_S = \Theta(n)$
- Speedup S is given by $S = \Theta(n / \log n)$



Performance Metrics: Speedup

- For a given problem, there might be **many serial algorithms** available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.
- For the purpose of computing speedup, we always consider **the best sequential program as the baseline**.

Performance Metrics: Speedup Example

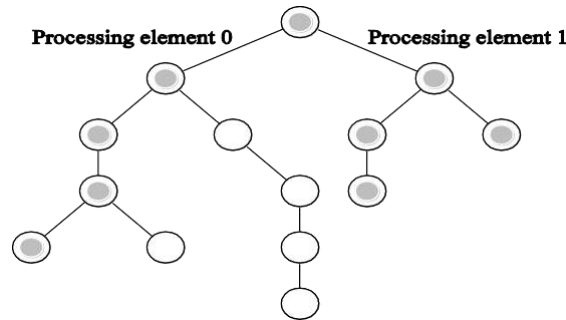
- Consider the problem of **parallel bubble sort**.
- The **serial time** for bubblesort is **150** seconds.
- The parallel time for **odd-even sort** (efficient parallelization of bubble sort) is **40** seconds.
- The speedup would appear to be $150/40 = 3.75$.
- But is this really a fair assessment of the system?
- What if **serial quicksort** only took 30 seconds? In this case, the speedup is $30/40 = 0.75$. This is a more realistic assessment of the system.

Performance Metrics: Speedup Bounds

- **Speedup can be as low as 0** (the parallel program never terminates).
- **Speedup, in theory, should be upper bounded by p** - after all, we can only expect a p -fold speedup if we use times as many resources.
- A **speedup greater than p is possible** only if each processing element spends less than time T_s/p solving the problem.
- In this case, a single processor could be timeslided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Superlinear Speedups

One reason for **superlinearity** is that the parallel version does less work than corresponding serial algorithm.



Searching an unstructured tree for a node with a given label, 'S', on two processing elements using **depth-first traversal**. The **two-processor** version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding **serial formulation expands the entire tree**. It is clear that **the serial algorithm does more work than the parallel algorithm**.

Performance Metrics: Efficiency

- **Efficiency** is a measure of the **fraction of time for which a processing element is usefully employed**
- Mathematically, it is given by

$$E = S / p = T_S / (p T_P) \quad (2)$$

- Following the bounds on speedup, efficiency can be **as low as 0 and as high as 1**.

Performance Metrics: Efficiency Example

- The speedup of **adding numbers** on processors is given by

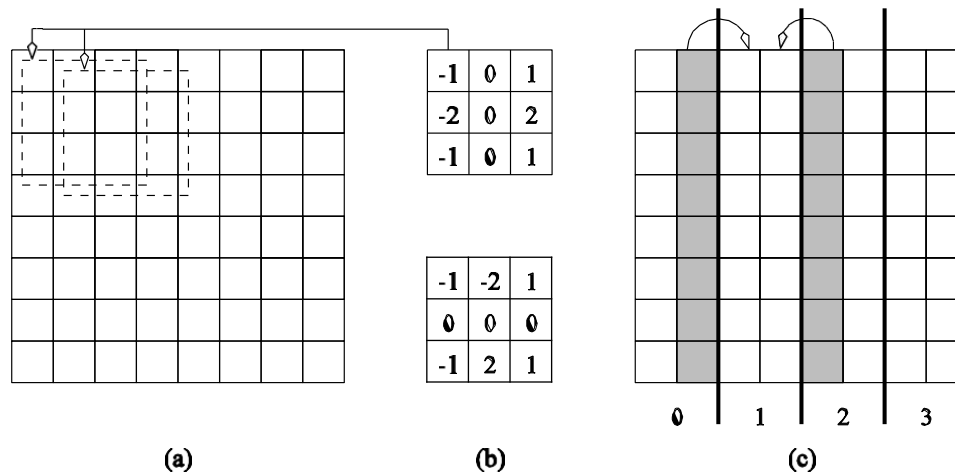
$$S = \frac{n}{\log n}$$

- Efficiency is given by

$$\begin{aligned} E &= \frac{\Theta\left(\frac{n}{\log n}\right)}{n} \\ &= \Theta\left(\frac{1}{\log n}\right) \end{aligned}$$

Parallel Time, Speedup, and Efficiency Example

Consider the problem of **edge-detection in images**. The problem requires us to apply a **3 x 3 template** to each pixel. If each multiply-add operation takes time t_c , the serial time for an $n \times n$ image is given by $T_s = 9t_c n^2$.



Example of edge detection: (a) an **8 x 8** image; (b) typical templates for detecting edges; and (c) partitioning of the image across **four processors** with **shaded regions indicating image data that must be communicated** from neighboring processors to processor 1.

Parallel Time, Speedup, and Efficiency Example (continued)

- One possible parallelization partitions the image equally into vertical segments, each with n^2 / p pixels.
- The boundary of each segment is $2n$ pixels. This is also the number of pixel values that will have to be communicated. This takes time $2(t_s + t_w n)$.
- Templates may now be applied to all n^2 / p pixels in time $9 t_c n^2 / p$.

Parallel Time, Speedup, and Efficiency Example (continued)

- The total time for the algorithm is therefore given by:

$$T_P = 9t_c \frac{n^2}{p} + 2(t_s + t_w n)$$

- The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c \frac{n^2}{p} + 2(t_s + t_w n)}$$

and

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

Cost of a Parallel System

- **Cost** is the product of parallel runtime and the number of processing elements used ($p \times T_P$).
- Cost reflects the sum of the **time** that each processing element **spends solving the problem**.
- A parallel system is said to be **cost-optimal** if the **cost of solving a problem on a parallel computer is asymptotically identical to serial cost**.
- Since $E = T_S / p T_P$, for cost optimal systems, $E = O(1)$.
- Cost is sometimes referred to as *work* or *processor-time product*.

Cost of a Parallel System: Example

Consider the problem of **adding numbers** on processors.

- We have, $T_p = \log n$ (for $p = n$).
- The cost of this system is given by $p T_p = n \log n$.
- Since the serial runtime of this operation is $\Theta(n)$, the algorithm is **not cost optimal**.

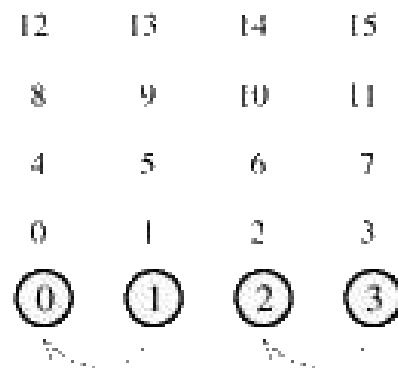
Effect of Granularity on Performance

- **Often, using fewer processors improves performance** of parallel systems.
- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called ***scaling down*** a parallel system.
- A **naive way of scaling down** is to think of each processor in the original case as a **virtual processor** and to assign virtual processors equally to scaled down processors.
- Since the number of processing elements decreases by a factor of n / p , the computation at each processing element increases by a factor of n / p .
- The communication cost should not increase by this factor since **some of the virtual processors assigned to a physical processors might talk to each other**. This is the basic reason for the improvement from building granularity.

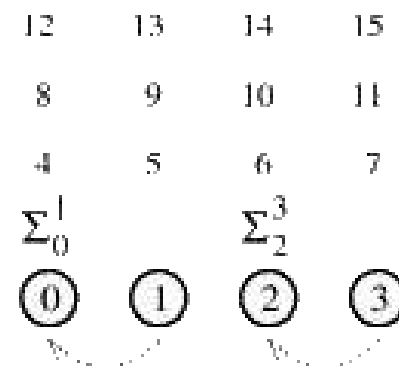
Building Granularity: Example

- Consider the problem of **adding n numbers** on **p** processing elements such that **$p < n$** and both **n** and **p** are powers of 2.
- Use the parallel algorithm for **n** processors, except, in this case, we think of them as **virtual processors**.
- Each of the **p** processors is now assigned **n / p** virtual processors.
- The first **$\log p$** of the **$\log n$** steps of the original algorithm are simulated in **$(n / p) \log p$** steps on **p** processing elements.
- Subsequent **$\log n - \log p$** steps **do not require any communication**.

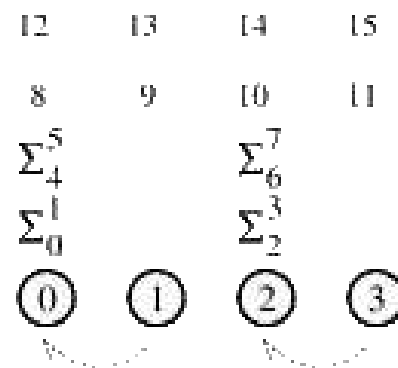
Building Granularity: Example (continued)



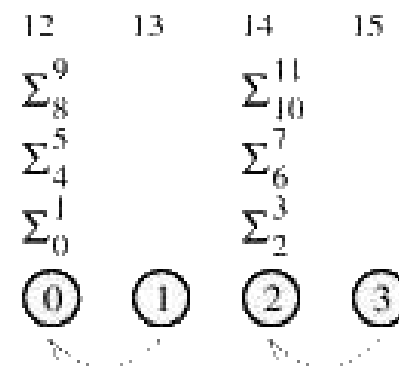
Substep 1



Substep 2



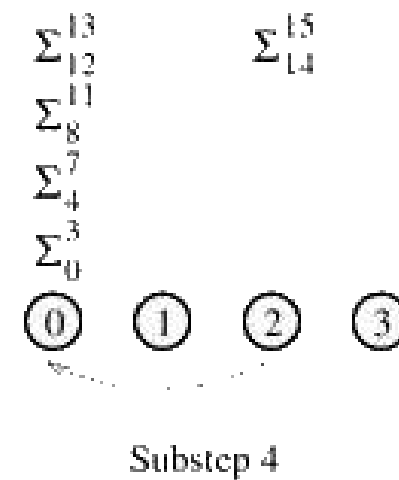
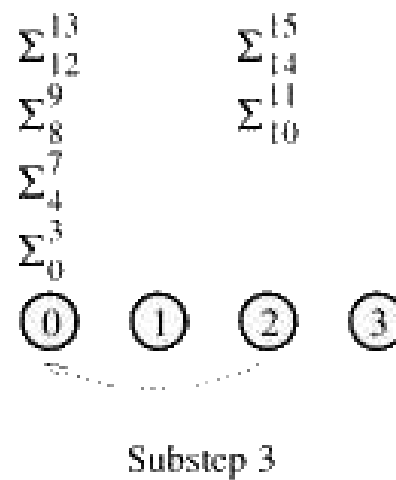
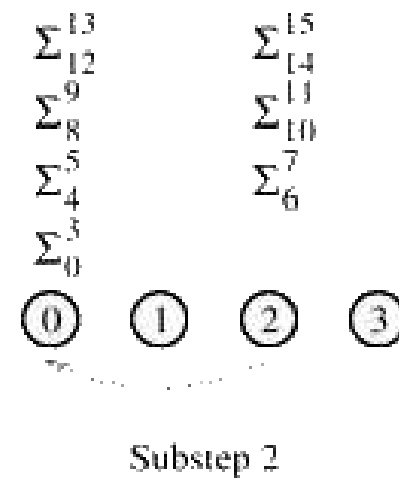
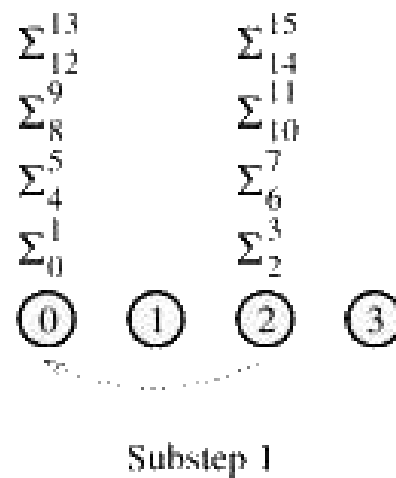
Substep 3



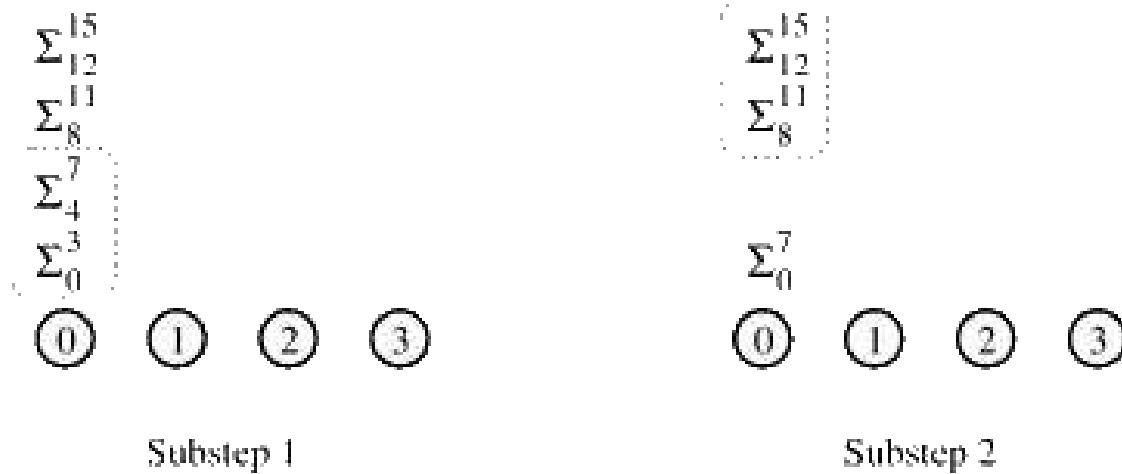
Substep 4

(a) Four processors simulating the first communication step of 16 processors

Building Granularity: Example (continued)



Building Granularity: Example (continued)



(c) Simulation of the third step in two substeps



(d) Simulation of the fourth step

(e) Final result

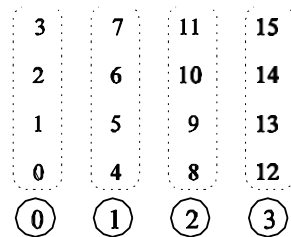
Building Granularity: Example (continued)

- The overall parallel execution time of this parallel system is $\Theta((n/p) \log p)$.
- The cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding n numbers sequentially. Therefore, the parallel system is **not cost-optimal**.

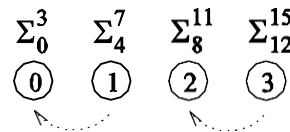
Building Granularity: Example (continued)

Can we build granularity in the example in a cost-optimal fashion?

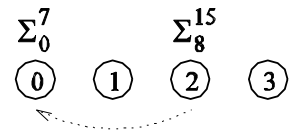
- Each processing element locally adds its n / p numbers in time $\Theta(n / p)$.
- The p partial sums on p processing elements can be added in time $\Theta(\log p)$.



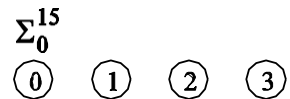
(a)



(b)



(c)



(d)

A cost-optimal way of computing the sum of 16 numbers using four processing elements.

Building Granularity: Example (continued)

- The parallel runtime of this algorithm is

$$T_P = \Theta(n/p + \log p), \quad (3)$$

- The cost is $\Theta(n + p \log p)$
- This is **cost-optimal**, so long as $n = \Omega(p \log p)$!

Programming Shared Address Space Platforms

Dr B Krishna Priya

Topic Overview

- Thread Basics
- The POSIX Thread API
- Synchronization Primitives in Pthreads
- Controlling Thread and Synchronization Attributes
- Composite Synchronization Constructs
- OpenMP: a Standard for Directive Based Parallel Programming

Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization.
- Process based models assume that all data associated with a process is private, by default, unless otherwise specified.
- Lightweight processes and threads assume that all memory is global.
- Directive based programming models extend the threaded model by facilitating creation and synchronization of threads.

Overview of Programming Models

- A *thread* is a single stream of control in the flow of a program. A program like:

```
for (row = 0; row < n; row++)  
    for (column = 0; column < n; column++)  
        c[row][column] =  
            dot_product(get_row(a, row),  
                        get_col(b, col));
```

can be transformed to:

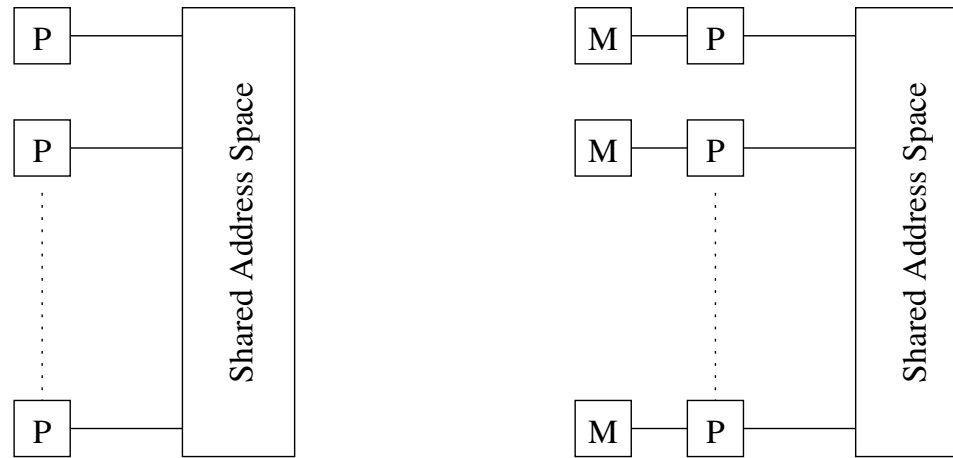
```
for (row = 0; row < n; row++)  
    for (column = 0; column < n; column++)  
        c[row][column] =  
            create_thread(  
                dot_product(get_row(a, row),  
                            get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.
- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.

Thread Basics



The logical machine model of a thread-based programming paradigm.

Thread Basics

- Threads provide software portability.
- Inherent support for latency hiding.
- Scheduling and load balancing.
- Ease of programming and widespread use.

The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

Thread Basics: Creation and Termination

- Pthreads provides two basic functions for specifying concurrency in a program:

```
#include <pthread.h>
int pthread_create (
    pthread_t    *thread_handle,
    const pthread_attr_t    *attribute,
    void *      (*thread_function)(void *),
    void      *arg);
```

```
int pthread_join (
    pthread_t    thread,
    void      **ptr);
```

- The function `pthread_create` invokes function `thread_function` as a thread.

Thread Basics: Creation and Termination (Example)

```
#include <pthread.h>
#include <stdlib.h>

#define MAX_THREADS      512
void *compute_pi (void *);
....

main() {
    ...
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;

    pthread_attr_init (&attr);
    for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
            (void *) &hits[i]);
    }
    for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}
```

Thread Basics: Creation and Termination (Example)

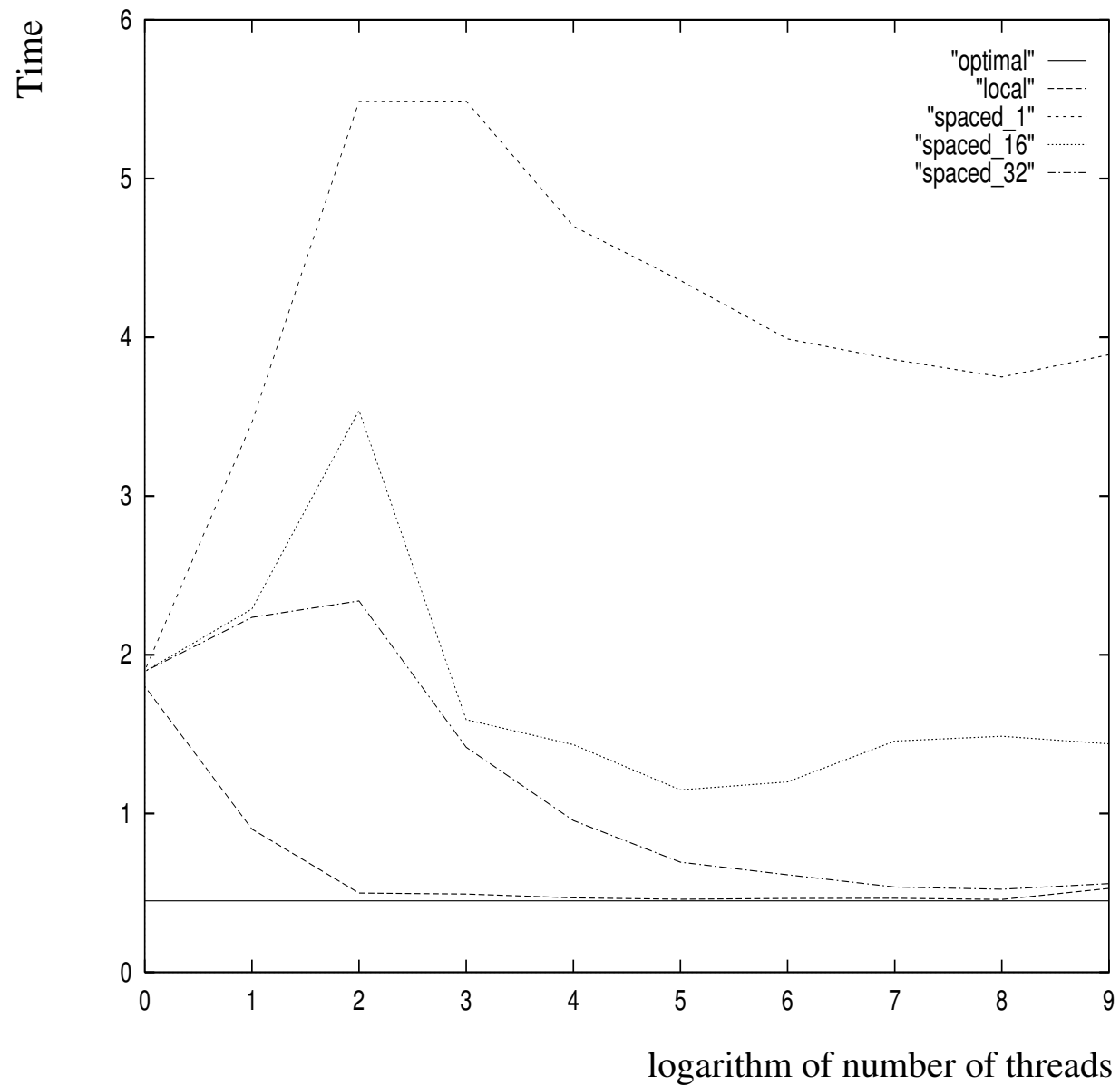
```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double rand_no_x, rand_no_y;
    int local_hits;

    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double) (rand_r(&seed)) / (double) ((2<<14)-1);
        rand_no_y = (double) (rand_r(&seed)) / (double) ((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
            (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
```

Programming and Performance Notes

- Note the use of the function `rand_r` (instead of superior random number generators such as `drand48`).
- Executing this on a 4-processor SGI Origin, we observe a 3.91 fold speedup at 32 threads. This corresponds to a parallel efficiency of 0.98!
- We can also modify the program slightly to observe the effect of false-sharing.
- The program can also be used to assess the secondary cache line size.

Programming and Performance Notes



Execution time of the `compute_pi` program.

Synchronization Primitives in Pthreads

- When multiple threads attempt to manipulate the same data item, the results can often be incoherent if proper care is not taken to synchronize them.

- Consider:

```
/* each thread tries to update variable best_cost as follows */  
if (my_cost < best_cost)  
    best_cost = my_cost;
```

- Assume that there are two threads, the initial value of `best_cost` is 100, and the values of `my_cost` are 50 and 75 at threads `t1` and `t2`.
- Depending on the schedule of the threads, the value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the threads.

Mutual Exclusion

- The code in the previous example corresponds to a critical segment; i.e., a segment that must be executed by only one thread at any time.
- Critical segments in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted.

Mutual Exclusion

The Pthreads API provides the following functions for handling mutex-locks:

```
int pthread_mutex_lock (  
    pthread_mutex_t    *mutex_lock);
```

```
int pthread_mutex_unlock (  
    pthread_mutex_t *mutex_lock);
```

```
int pthread_mutex_init (  
    pthread_mutex_t    *mutex_lock,  
    const pthread_mutexattr_t    *lock_attr);
```

Mutual Exclusion

We can now write our previously incorrect code segment as:

```
pthread_mutex_t minimum_value_lock;
...
main() {
    ....
    pthread_mutex_init(&minimum_value_lock, NULL);
    ....
}

void *find_min(void *list_ptr) {
    ....
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
        minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
}
```


Producer-Consumer Using Locks

The producer-consumer scenario imposes the following constraints:

- The producer thread must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads must not pick up tasks until there is something present in the shared data structure.
- Individual consumer threads should pick up tasks one at a time.

Producer-Consumer Using Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
...
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}

void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

Producer-Consumer Using Locks

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Types of Mutexes

- Pthreads supports three types of mutexes – normal, recursive, and error-check.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex allows a single thread to lock a mutex as many times as it wants. It simply increments a count on the number of locks. A lock is relinquished by a thread when the count becomes zero.
- An error check mutex reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).
- The type of the mutex can be set in the attributes object before it is passed at time of initialization.

Overheads of Locking

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, `pthread_mutex_trylock`.

```
int pthread_mutex_trylock (  
    pthread_mutex_t *mutex_lock);
```

- `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list */
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}

int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

Alleviating Locking Overhead (Example)

```
/* rewritten output_record function */

int output_record(struct database_record *record_ptr) {
    int count;
    int lock_status;
    lock_status = pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
                      requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

Condition Variables for Synchronization

- A condition variable allows a thread to block itself until specified data reaches a predefined state.
- A condition variable is associated with this predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- A single condition variable may be associated with more than one predicate.
- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.
- If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`.

Condition Variables for Synchronization

Pthreads provides the following functions for condition variables:

```
int pthread_cond_wait(pthread_cond_t *cond,  
    pthread_mutex_t *mutex);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_init(pthread_cond_t *cond,  
    const pthread_condattr_t *attr);
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;

/* other data structures here */

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

Producer-Consumer Using Condition Variables

```
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                             &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

Producer-Consumer Using Condition Variables

```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                              &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using *attributes objects*.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions:
`pthread_attr_setdetachstate,`
`pthread_attr_setguardsize_np,`
`pthread_attr_setstacksize,`
`pthread_attr_setinheritsched,`
`pthread_attr_setschedpolicy,` and
`pthread_attr_setschedparam.`

Attributes Objects for Mutexes

- Initialize the attributes object using function: `pthread_mutexattr_init`.
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.

```
pthread_mutexattr_settype_np (  
    pthread_mutexattr_t    *attr,  
    int    type);
```

Here, `type` specifies the type of the mutex and can take one of:

- `PTHREAD_MUTEX_NORMAL_NP`
- `PTHREAD_MUTEX_RECURSIVE_NP`
- `PTHREAD_MUTEX_ERRORCHECK_NP`

Composite Synchronization Constructs

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs – read-write locks and barriers.

Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
- A read lock is granted when there are other threads that may already have read locks.
- If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
- If there are multiple threads requesting a write lock, they must perform a condition wait.
- With this description, we can design functions for read locks `mylib_rwlock_rlock`, write locks `mylib_rwlock_wlock`, and unlocking `mylib_rwlock_unlock`.

Read-Write Locks

- The lock data type `mylib_rwlock_t` holds the following:
 - a count of the number of readers,
 - the writer (a 0/1 integer specifying whether a writer is present),
 - a condition variable `readers_proceed` that is signaled when readers can proceed,
 - a condition variable `writer_proceed` that is signaled when one of the writers can proceed,
 - a count `pending_writers` of pending writers, and
 - a mutex `read_write_lock` associated with the shared data structure.

Read-Write Locks

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;

void mylib_rwlock_init (mylib_rwlock_t *l) {
    l -> readers = 1 -> writer = 1 -> pending_writers = 0;
    pthread_mutex_init(&(l -> read_write_lock), NULL);
    pthread_cond_init(&(l -> readers_proceed), NULL);
    pthread_cond_init(&(l -> writer_proceed), NULL);
}
```

Read-Write Locks

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    /* if there is a write lock or pending writers, perform condition
       wait.. else increment count of readers and grant read lock */

    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> pending_writers > 0) || (l -> writer > 0))
        pthread_cond_wait(&(l -> readers_proceed),
                           &(l -> read_write_lock));
    l -> readers ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
    /* if there are readers or writers, increment pending writers
       count and wait. On being woken, decrement pending writers
       count and increment writer count */

    pthread_mutex_lock(&(l -> read_write_lock));
    while ((l -> writer > 0) || (l -> readers > 0)) {
        l -> pending_writers ++;
        pthread_cond_wait(&(l -> writer_proceed),
            &(l -> read_write_lock));
    }
    l -> pending_writers --;
    l -> writer ++;
    pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Read-Write Locks

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
    /* if there is a write lock then unlock, else if there are
       read locks, decrement count of read locks. If the count
       is 0 and there is a pending writer, let it through, else
       if there are pending readers, let them all go through */

    pthread_mutex_lock(&(l -> read_write_lock));
    if (l -> writer > 0)
        l -> writer = 0;
    else if (l -> readers > 0)
        l -> readers--;
    pthread_mutex_unlock(&(l -> read_write_lock));
    if ((l -> readers == 0) && (l -> pending_writers > 0))
        pthread_cond_signal(&(l -> writer_proceed));
    else if (l -> readers > 0)
        pthread_cond_broadcast(&(l -> readers_proceed));
}
```

Barriers

- As in MPI, a barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

Barriers

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

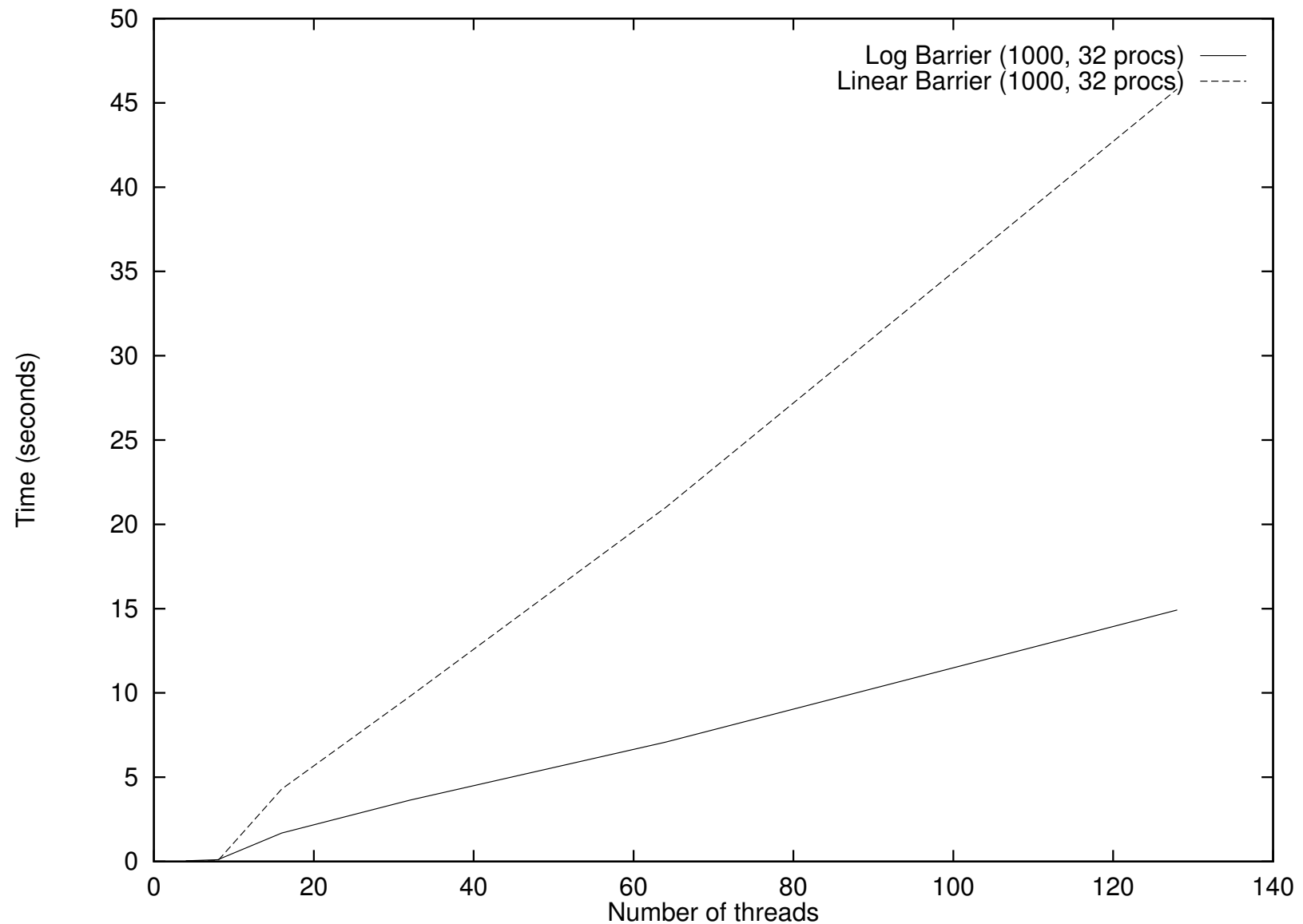

Barriers

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b -> ok_to_proceed),
                                &(b -> count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```

Barriers

- The barrier described above is called a linear barrier.
- The trivial lower bound on execution time of this function is therefore $O(n)$ for n threads.
- This implementation of a barrier can be speeded up using multiple barrier variables organized in a tree.
- We use $n/2$ condition variable-mutex pairs for implementing a barrier for n threads.
- At the lowest level, threads are paired up and each pair of threads shares a single condition variable-mutex pair.
- Once both threads arrive, one of the two moves on, the other one waits.
- This process repeats up the tree.
- This is also called a log barrier and its runtime grows as $O(\log p)$.

Barrier



Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

Tips for Designing Asynchronous Programs

- Never rely on scheduling assumptions when exchanging data.
- Never rely on liveness of data resulting from assumptions on scheduling.
- Do not rely on scheduling as a means of synchronization.
- Where possible, define and use group synchronizations and data replication.

Overview of Programming Models

- Programming models provide support for expressing concurrency and synchronization.
- **Process based models** assume that all data associated with a process is private, by default, unless otherwise specified.
- **Lightweight processes and threads** assume that all memory is global.
- **Directive based programming models** extend the threaded model by facilitating creation and synchronization of threads.

What is a Thread?

- A thread is defined as an independent stream of instructions that can be packaged and executed by the operating system (similar to a process but much less overhead) .
- Multiple threads can be executed in parallel on many computer systems. This *multithreading* generally occurs by **time slicing** in which case the processing is not literally simultaneous, for the single processor is really doing only one thing at a time.
- Unlike processes, threads typically share the state information of a single process, and share memory and other resources directly.

Overview of Programming Models

- A *thread* is a single stream of control in the flow of a program. A program like:

```
for (row = 0; row < n; row++)  
    for (col = 0; col < n; col++)  
        c[row][col] =  
            dot_product(get_row(a, row),  
                        get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)  
    for (col = 0; col < n; col++)  
        c[row][col] =  
            create_thread(dot_product(get_row(a, row),  
                                    get_col(b, col)));
```

In this case, one may think of the thread as an instance of a function that returns before the function has finished executing.

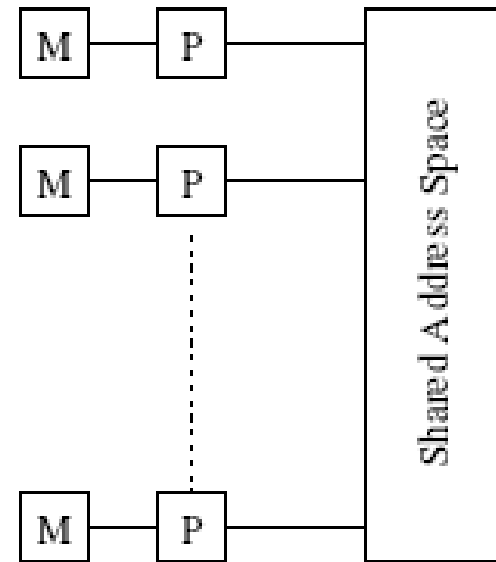
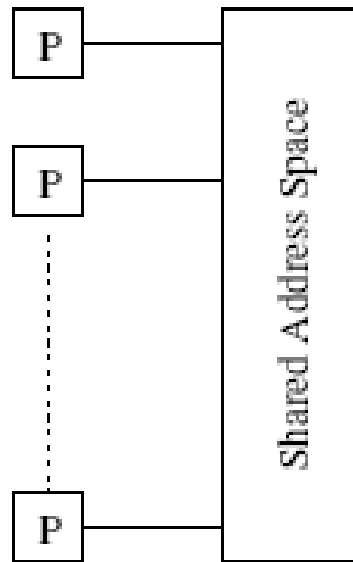
Thread Basics

- Threads provide software portability.
- Inherent support for latency hiding.
- Scheduling and load balancing.
- Ease of programming and widespread use.

Thread Basics

- All memory in the logical machine model of a thread is globally accessible to every thread.
- The stack corresponding to the function call is generally treated as being local to the thread for liveness reasons.
- This implies a logical machine model with both global memory (default) and local memory (stacks).
- It is important to note that such a flat model may result in very poor performance since memory is physically distributed in typical machines.

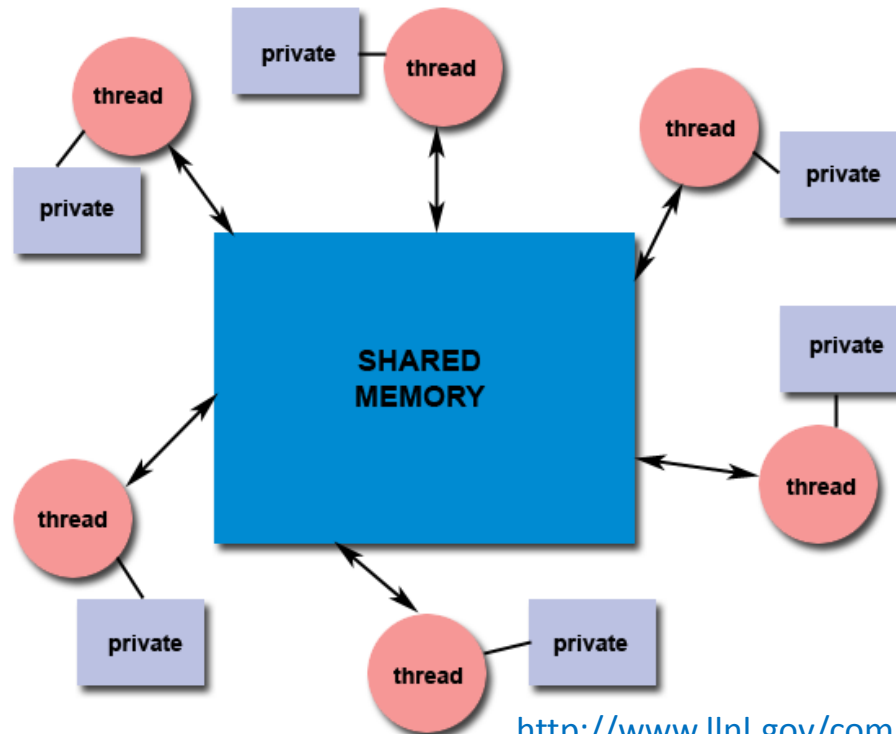
Thread Basics



- The logical machine model of a thread-based programming paradigm.

Shared Memory Model

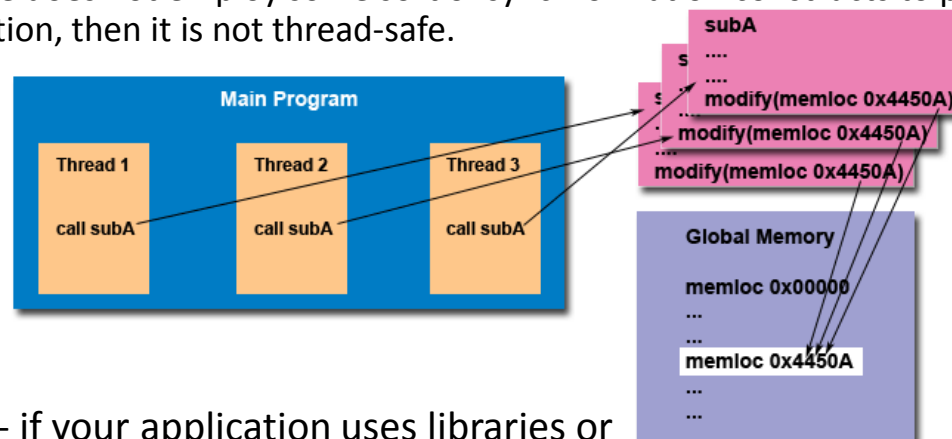
- All threads have access to the same global / shared memory.
- Additionally, threads have their own private data.
- Programmers are responsible for synchronizing access to global memory.



<http://www.llnl.gov/computing/tutorials/threads/>

Thread Safeness

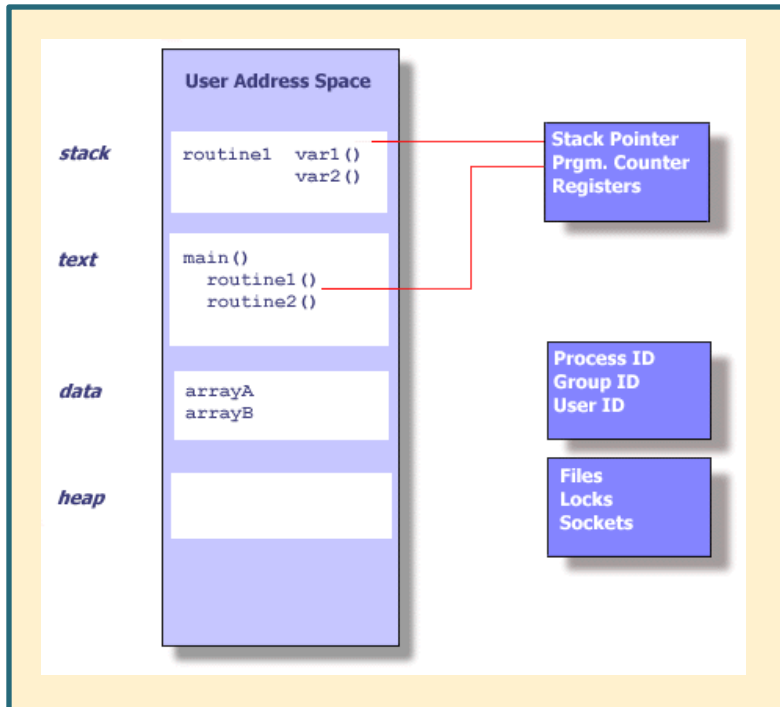
- Thread Safeness refers to the ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose we have an application which creates three threads each of which calls subA(). Additionally:
 - subA() accesses/modifies a global structure or location in memory.
 - As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
 - If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.



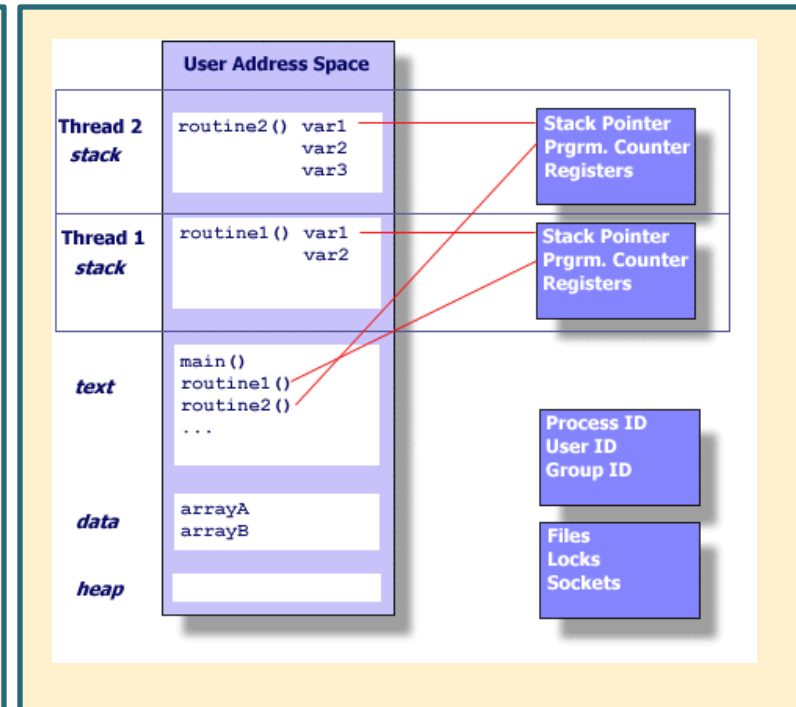
- **BE CAREFUL!** - if your application uses libraries or other objects that don't explicitly guarantee thread-safeness – assume that they are **NOT** thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine.

<http://www.llnl.gov/computing/tutorials/pthreads/>

POSIX Threads



Standard Unix Process



Unix Process w/Threads

<http://www.llnl.gov/computing/tutorials/pthreads/>

Threads duplicate on the bare essential resources that enable them to exist as executable code.

POSIX Threads

- Independent flow of control is achieved by having each thread maintain its own:
 - Stack Pointer
 - Registers
 - Scheduling Properties (priority, etc.)
 - Signals
 - Thread Specific Data

<http://www.llnl.gov/computing/tutorials/pthreads/>

POSIX Threads

- Also, because threads share resources within the same process
 - Changes made by one thread to shared resources, i.e., closing a file, will be visible to all other threads.
 - Pointers with the same addresses point to the same data.
 - Explicit synchronization must be performed by the programming when accessing (reading/writing) the same memory (race conditions).

<http://www.llnl.gov/computing/tutorials/pthreads/>

The POSIX Thread API

Pthreads

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations which adhere to this standard are referred to as POSIX threads, or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's.

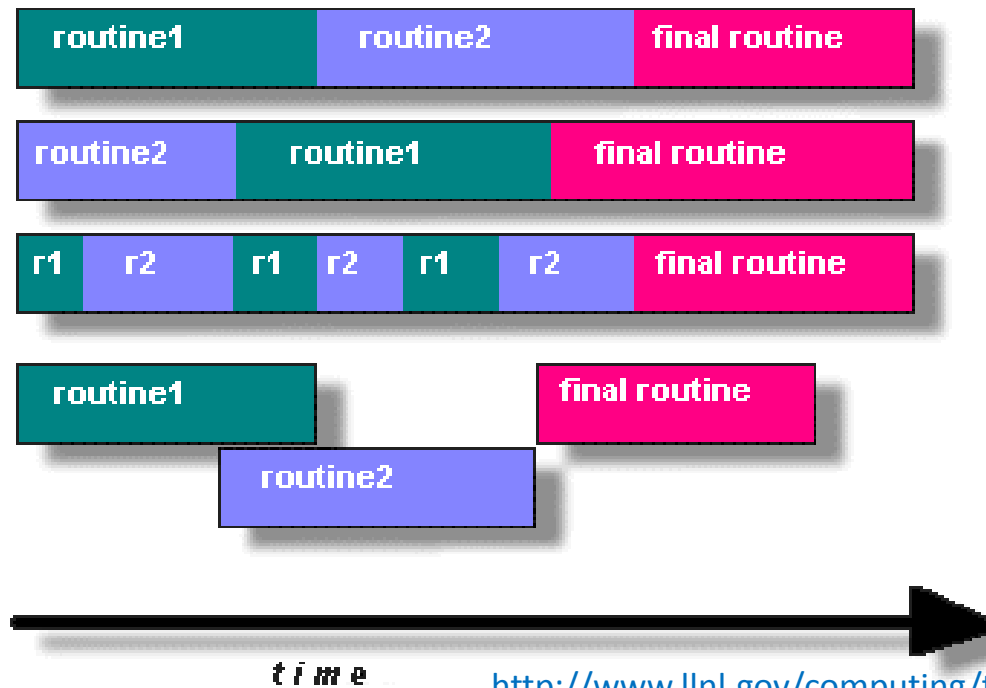
<http://www.llnl.gov/computing/tutorials/pthreads/>

POSIX Threads (Pthreads)

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a “***pthread.h***” header/include file and a thread library - though the this library may be part of another library, such as libc.

Why Pthreads?

- In general though, in order for a program to take advantage of Pthreads, it must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.



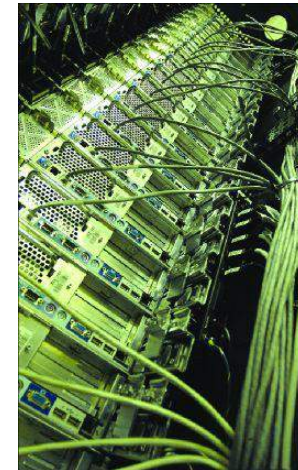
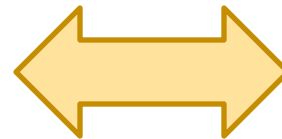
<http://www.llnl.gov/computing/tutorials/threads/>

Why Pthreads?

- The primary motivation for using Pthreads is to realize potential program performance gains.
- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- On modern, multi-cpu machines, Pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel Pthreads programs.



Intel Quad Core
(Threads)



Cluster
(MPI)

Pthreads API

- The Pthreads API can be informally grouped into three major classes:
 1. **Thread Management**: functions that work directly on threads, i.e., creating, detaching, joining, etc. These include functions to set/query thread attributes (joinable, scheduling etc.)
 2. **Mutexes**: functions that deal with synchronization, called a "mutex", which include functions for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
 3. **Condition Variables**: functions that address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Naming Conventions

- All functions in the thread library begin with **pthread_**

Routine Prefix	Functional Group
pthread_	Threads themselves and misc subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

- The Pthreads API contains over 60 routines.
- For portability, the ***pthread.h*** header file should be included in each source file using the Pthreads library.

<http://www.llnl.gov/computing/tutorials/pthreads/>

Compiling Threaded Programs

- All functions in the thread library begin with **pthread_**

Compiler / Platform	Compiler Command	Description
IBM AIX	xlc_r / cc_r	C (ANSI / non-ANSI)
	xlc_r	C++
	xlf_r -qnosave xlf90_r -qnosave	Fortran - using IBM's Pthreads API
INTEL Linux	icc -pthread	C
	icpc -pthread	C++
PathScale Linux	pathcc -pthread	C
	pathCC -pthread	C++
PGI Linux	pgcc -lpthread	C
	pgCC -lpthread	C++
GNU Linux, AIX	gcc -pthread	GNU C
	g++ -pthread	GNU C++

<http://www.llnl.gov/computing/tutorials/pthreads/>

Creating Threads

- Routines:
 - **pthread_create** (thread, attr, start_routine, arg)
 - **pthread_attr_init** (attr)
 - **pthread_attr_destroy** (attr)
- pthread_create arguments:
 - **thread**: an opaque, unique identifier for the new thread returned by the subroutine.
 - **attr**: an opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
 - **start_routine**: the C routine that the thread will execute once it is created.
 - **arg**: a single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

<http://www.llnl.gov/computing/tutorials/pthreads/>

Creating Threads

- Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- **`pthread_create`** - creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- The maximum number of threads that may be created by a process is implementation dependent.
- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.

Terminating Threads

- There are several ways in which a Pthread may be terminated:
 - The thread returns from its starting routine (the main routine for the initial thread).
 - The thread makes a call to the **pthread_exit()** routine.
 - The thread is cancelled by another thread via the **pthread_cancel()** routine.
 - The entire process is terminated due to a call to either the exec or exit routines.
- **pthread_exit()** is used to explicitly exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work and is no longer required to exist.

Terminating Threads

- If `main()` finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()` finishes.
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread.
- Cleanup: the `pthread_exit()` routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Example: Thread Creation

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid)
{
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread #%d!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

- This simple example code creates 5 threads with the **pthread_create()** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **pthread_exit()**.
<http://www.llnl.gov/computing/tutorials/threads/>

Example: Argument Passing

- The **pthread_create()** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the **pthread_create()** routine.
- All arguments must be passed by reference and cast to (**void ***).

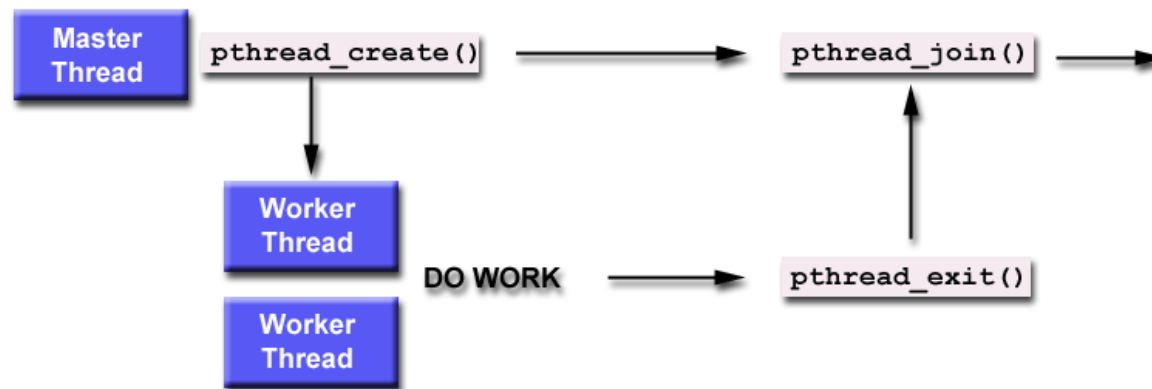
```
int *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
    taskids[t] = (int *) malloc(sizeof(int));
    *taskids[t] = t;
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) taskids[t]);
    ...
}
```

<http://www.llnl.gov/computing/tutorials/threads/>

Thread Management

- Routines:
 - **pthread_join**(threadid,status)
 - **pthread_detach**(threadid,status)
 - **pthread_attr_setdetachstate**(attr,detachstate)
 - **pthread_attr_getdetachstate**(attr,detachstate)
- "Joining" is one way to accomplish synchronization between threads. For example:



- The **pthread_join()** subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to **pthread_exit()**.
- A joining thread can match one **pthread_join()** call. It is a logical error to attempt multiple joins on the same thread.

<http://www.llnl.gov/computing/tutorials/pthreads/>

Joining and Detaching Threads

Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable. However, not all implementations may follow this.
- To explicitly create a thread as joinable or detached, the **attr** argument in the **pthread_create()** routine is used. The typical 4 step process is:
 - Declare a pthread attribute variable of the **pthread_attr_t** data type
 - Initialize the attribute variable with **pthread_attr_init()**
 - Set the attribute detached status with **pthread_attr_setdetachstate()**
 - When done, free library resources used by the attribute with **pthread_attr_destroy()**

Detaching:

- The **pthread_detach()** routine can be used to explicitly detach a thread even though it was created as joinable.
- There is no converse routine.

<http://www.llnl.gov/computing/tutorials/pthreads/>

Stack Management

Routines

- **pthread_attr_getstacksize**(attr, stacksize)
- **pthread_attr_setstacksize**(attr, stacksize)
- **pthread_attr_getstackaddr**(attr, stackaddr)
- **pthread_attr_setstackaddr**(attr, stackaddr)

Preventing Stack Problems:

- The POSIX standard does not dictate the size of a thread's stack. This is implementation dependent and varies.
- Exceeding the default stack limit is often very easy to do, with the usual results: program termination and/or corrupted data.
- Safe and portable programs do not depend upon the default stack limit, but instead, explicitly allocate enough stack for each thread by using the **pthread_attr_setstacksize** routine.
- The **pthread_attr_getstackaddr** and **pthread_attr_setstackaddr** routines can be used by applications in an environment where the stack for a thread must be placed in some particular region of memory.

<http://www.llnl.gov/computing/tutorials/threads/>

Stack Management Example

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4
#define N 1000
#define MEGEXTRA 1000000

pthread_attr_t attr;

void *dowork(void *threadid)
{
    double A[N][N];
    int i,j;
    size_t mystacksize;

    pthread_attr_getstacksize (&attr, &mystacksize);
    printf("Thread d%: stack size = %li bytes \n", threadid,
        mystacksize);
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i][j] = ((i*j)/3.452) + (N-i);
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
    pthread_t threads[NTHREADS];
    size_t stacksize;
    int rc, t;

    pthread_attr_init(&attr);
    pthread_attr_getstacksize (&attr, &stacksize);
    printf("Default stack size = %li\n", stacksize);
    stacksize = sizeof(double)*N*N+MEGEXTRA;
    printf("Amount of stack needed per thread = %li\n",stacksize);
    pthread_attr_setstacksize (&attr, stacksize);
    printf("Creating threads with stack size = %li bytes\n",stacksize);
    for(t=0; t<NTHREADS; t++){
        rc = pthread_create(&threads[t], &attr, dowork, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    printf("Created %d threads.\n", t);
    pthread_exit(NULL);
}
```

<http://www.llnl.gov/computing/tutorials/pthreads/>