What is concurrency?

- What is a sequential program?
  - A single thread of control that executes one instruction and when it is finished execute the next logical instruction
- What is a concurrent program?
  - A collection of autonomous sequential threads, executing (logically) in parallel
- The implementation (i.e. execution) of a collection of threads can be:

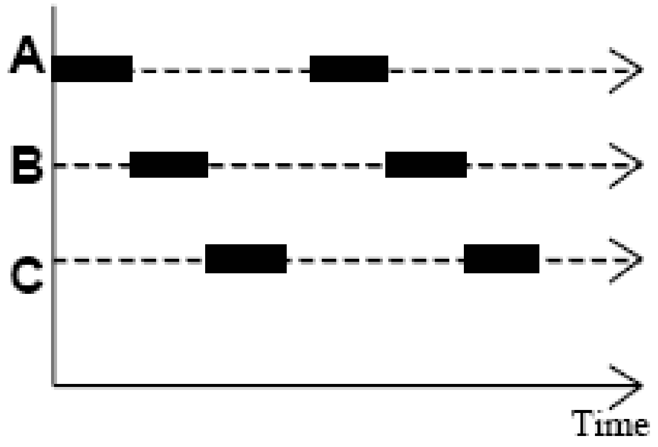
  Multiprogramming
  - Threads multiplex their executions on a single processor.

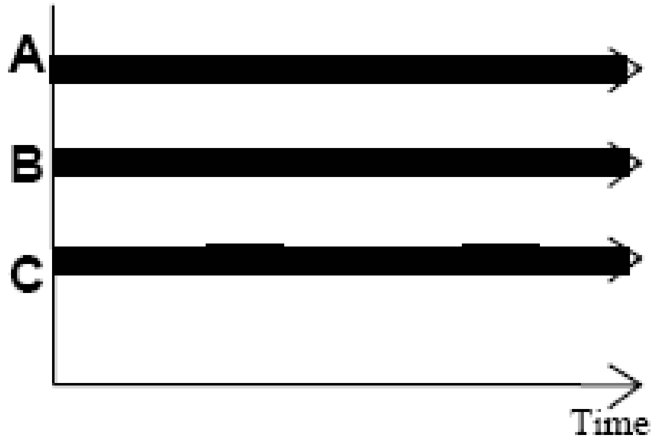
  Multiprocessing
  - Threads multiplex their executions on a multiprocessor or a multicore system

  Distributed Processing
  - Processes multiplex their executions on several different machines

Concurrency and Parallelism

- Concurrency is not (only) parallelism
- Interleaved Concurrency
  - Logically simultaneous processing
  - Interleaved execution on a single processor
- Parallelism
  - Physically simultaneous processing
  - Requires a multiprocessors or a multicore system





Synchronization

- All the interleavings of the threads are NOT acceptable correct programs.
- Java provides synchronization mechanism to restrict the interleavings
- Synchronization serves two purposes:
  - **Ensure safety** for shared updates
    - Avoid **race conditions**
  - **Coordinate** actions of threads
    - Parallel computation
    - Event notification

Safety

- Multiple threads access shared resource simultaneously
- **Safe** only if:
  - All accesses have no effect on resource,
    - e.g., reading a variable,

  or

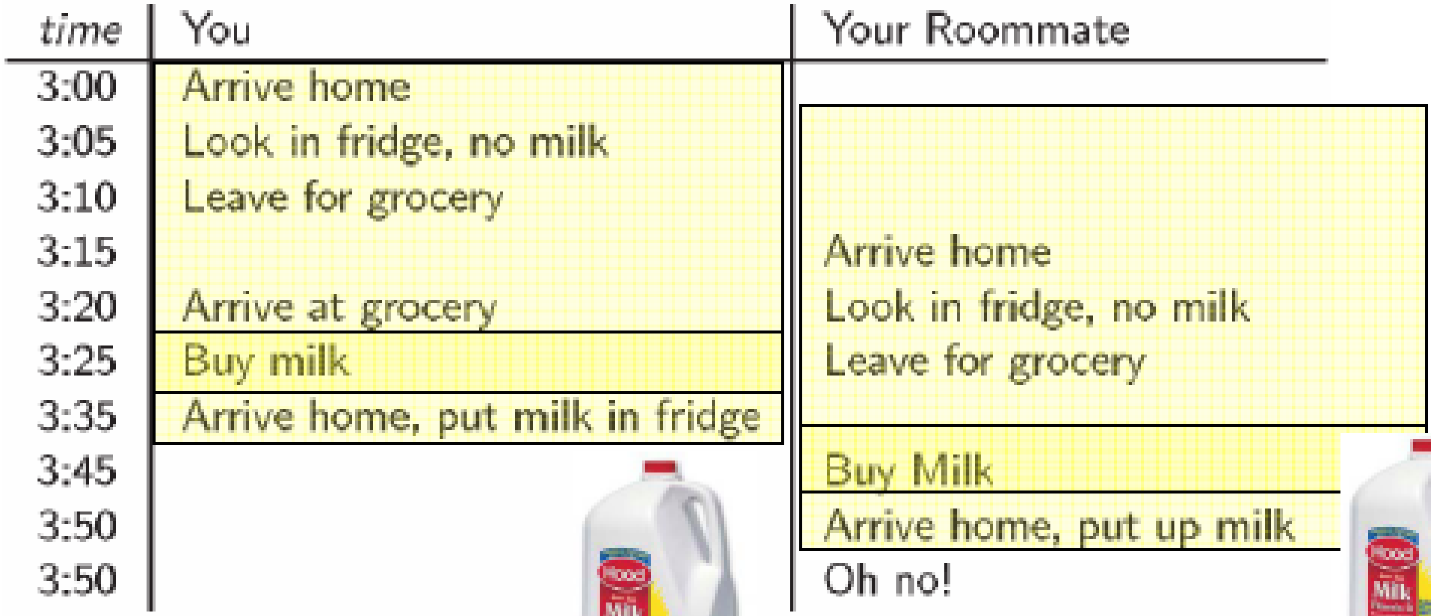
  - All accesses *idempotent*
    - E.g., `y = sign(a)`, `a = a*2;`

  or

  - Only one access at a time: *mutual exclusion*

Safety: Example

- "The *too much milk* problem"



time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home, put milk in fridge	
3:45		Buy Milk
3:50		Arrive home, put up milk
3:50		Oh no!

- Model of need to **synchronize** activities

Why You Need Locks

thread A

```
if (no milk && no note)
    leave note
    buy milk
    remove note
```

thread B

```
if (no milk && no note)
    leave note
    buy milk
    remove note
```

too much milk

- Does this w

Mutual Exclusion

- Prevent more than one thread from accessing *critical section* at a given time
  - Once a thread is in the critical section, no other thread can enter that critical section until the first thread has left the critical section.
  - No interleavings of threads within the critical section
  - **Serializes** access to section

```
synchronized int getbal() {
        return balance;
  }

synchronized void post(int v) {
        balance = balance + v;
  }
```

Principle of Operation

One of the most popular ways of explaining how pipeline works is using the process of doing laundry. The laundry process consists of three standalone operations performed on the clothes:

1. Washing
2. Drying
3. Folding

The washing is performed using a washing machine, the drying using a dryer machine and the folding is performed manually by the person doing the laundry. For the purpose of the example let’s say the washing takes **60 minutes**, the drying takes **30 minutes** and the folding of the clothes takes **30 minutes**. As each operation starts only after the previous has finished we have one load of laundry completed for **2 hours**.

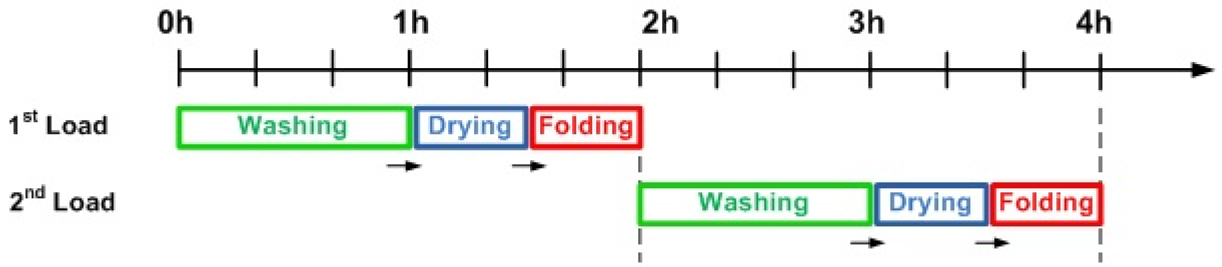


Fig. 1 Laundry process without pipeline technique

In Fig. 1 we can see the timeline of these operations. If we look closely at the process, one thing becomes obvious – once an operation is finished for the current load of clothes, the hardware used stays idle and waits for the next load. For example, the washer machine is idle while drying and folding operations are performed. This is certainly not the most efficient way of doing things and one way to improve it is shown in Fig. 2. There we can see the same process of doing laundry, but this time using a pipeline technique. As soon as the washing is completed, we can put the clothes of the 2nd load, so the washing machine keeps working while the drying and the folding of the 1st load are being executed.

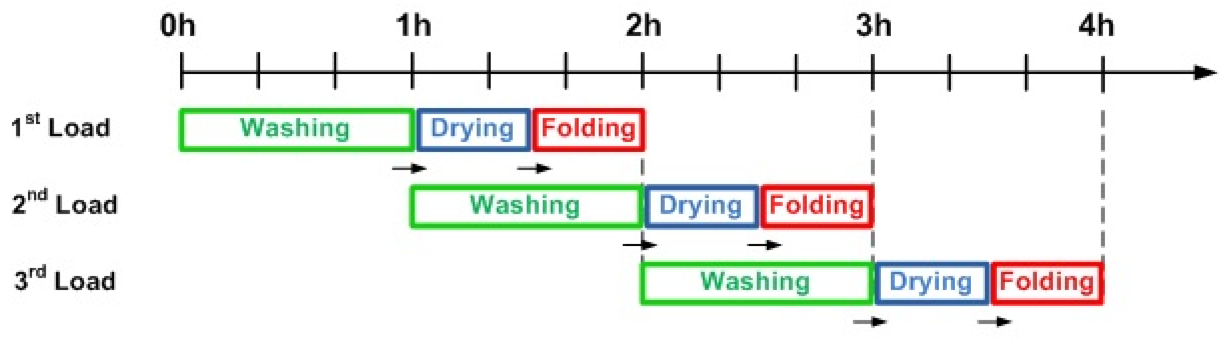


Fig. 2 Laundry process with pipeline technique

Based on the examples above, we can conclude that using the pipeline technique does not have an impact on the time needed for completing a single load of laundry (it takes 2 hours). The improvement is visible when doing multiple loads. Without the pipeline, we can do two loads for a total of 4 hours. Using the pipeline as shown in Fig. 2 we can do 3 loads in the same time frame. The speedup would be even greater if all of the operations took the same time to complete, thus allowing better overlapping in the pipeline. This is applied in microprocessor pipeline implementations.

Microprocessor Pipeline Example

Now let's see how the pipeline technique is applied to a microprocessor.
It should be noted that there are many different pipeline implementations and each can have a different number of pipeline stages. For this conceptual example, we will keep things simple and use RISC load and store CPU architecture with 5 stage pipeline. The stages are:

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Writeback (WB)

Each stage is executed by its own dedicated CPU functional unit and each takes one clock cycle to execute.

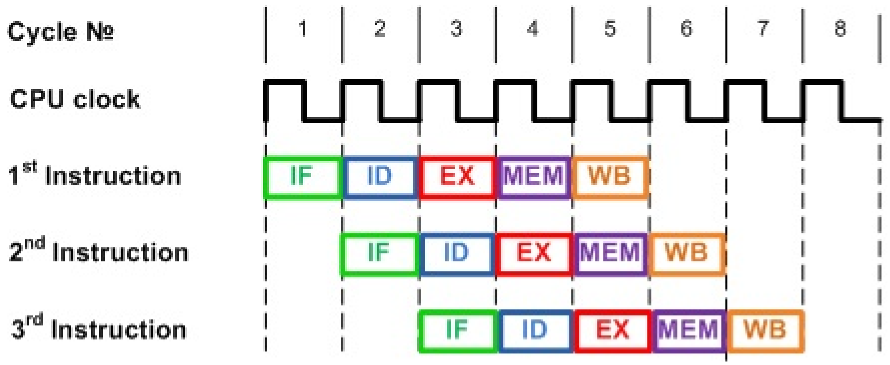


Fig. 3 Microprocessor instruction pipeline

In Fig. 3 we can see how instructions are overlapped using the pipelining technique. In the first clock cycle, the first instruction is fetched. In the following clock cycle, that same instruction is decoded while at the same time, the second instruction is being fetched. During the third clock cycle, the first instruction is executed, the second instruction is decoded and the third instruction is being fetched. On the fifth clock cycle, we have the first instruction completed. From then on, we have an instruction completing on every clock cycle. The time for reaching the completion of the first instruction passed through the pipeline is called “**time to fill**“. It is dependent on the number of stages. In this example, it takes 5 cycles to fill the pipeline.

Performance Improvement

In order to properly identify the performance improvement that can be achieved using the pipelining technique, we need to use two terms – **latency** and **throughput**.

Latency *is the time it takes for an operation to complete.*

Throughput *is the number of operations that are completed in a certain time frame.*

Looking at the example shown in Fig. 3 we can make the following conclusions:

- The pipeline does not affect the latency. The instruction cycle consisting of 5 phases is the same whether pipelining is used or not. Each instruction takes 5 clock cycles to complete.
- The pipeline improves the throughput, with potential speedup equal to the number of pipeline stages.

Once the pipeline is filled and continuously fed, we can have an instruction completed on every clock cycle (**C**ycles **P**er **I**nstruction (**CPI**) = 1). This is, of course, the ideal case not taking into account some of the pipeline limitations mentioned in the next chapter.

Pipeline Limitations

There are some known limitations to pipelining technique used in microprocessors. Hazards prevent the next instruction in the pipeline to be executed in its designated clock cycle. The pipeline hazards can be grouped into three categories:

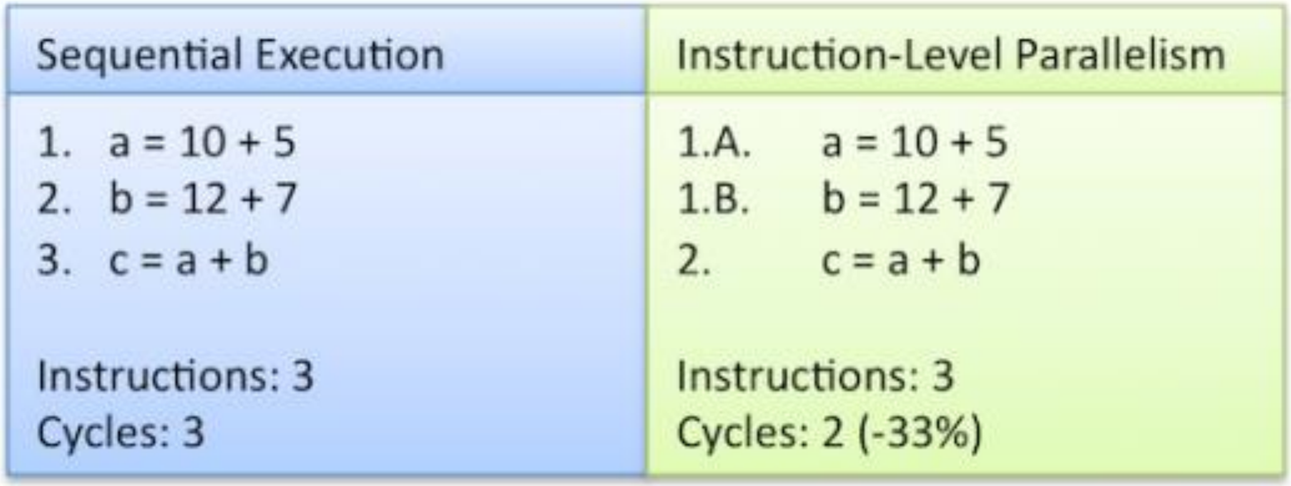
- Data Hazard – Can occur when an instruction depends on the result of a previous instruction still being processed in the pipeline.
- Structural Hazard – Can occur when the available hardware does not support some instruction combinations.
- Control Hazards – Caused by jump and branch instructions. These instructions will usually require the flushing (emptying) of the pipeline and loading it with instructions from addresses pointed by the branch and jump instructions.

2.1 Instruction-Level Parallelism

The first key type of parallelism, instruction-level parallelism (or ILP), involves executing certain instructions of a program simultaneously which would otherwise be executed sequentially [Goossens10], which may positively impact performance depending on the instruction mix in the application.

Most modern CPUs utilize instruction-level parallelization techniques such as pipelining, superscalar execution, prediction, out-of-order execution, dynamic branch prediction or address speculation [Goossens10]. However, only certain portions of a given program's instruction set may be suitable for instruction-level parallelization, as a simple example illustrates below in Figure 2. Because steps 1 and 2 of the sequential operation are independent of each other, a processor employing instruction-level parallelism can run instructions 1.A. and 1.B. simultaneously and thereby reduce the operation cycles to complete the operation by 33%. The last step must be executed sequentially in either case, however, as it is dependent on the two prior steps.

Example:
Note: This example is an oversimplification, but it generally conveys both the potential benefit and potential limits of the technique.



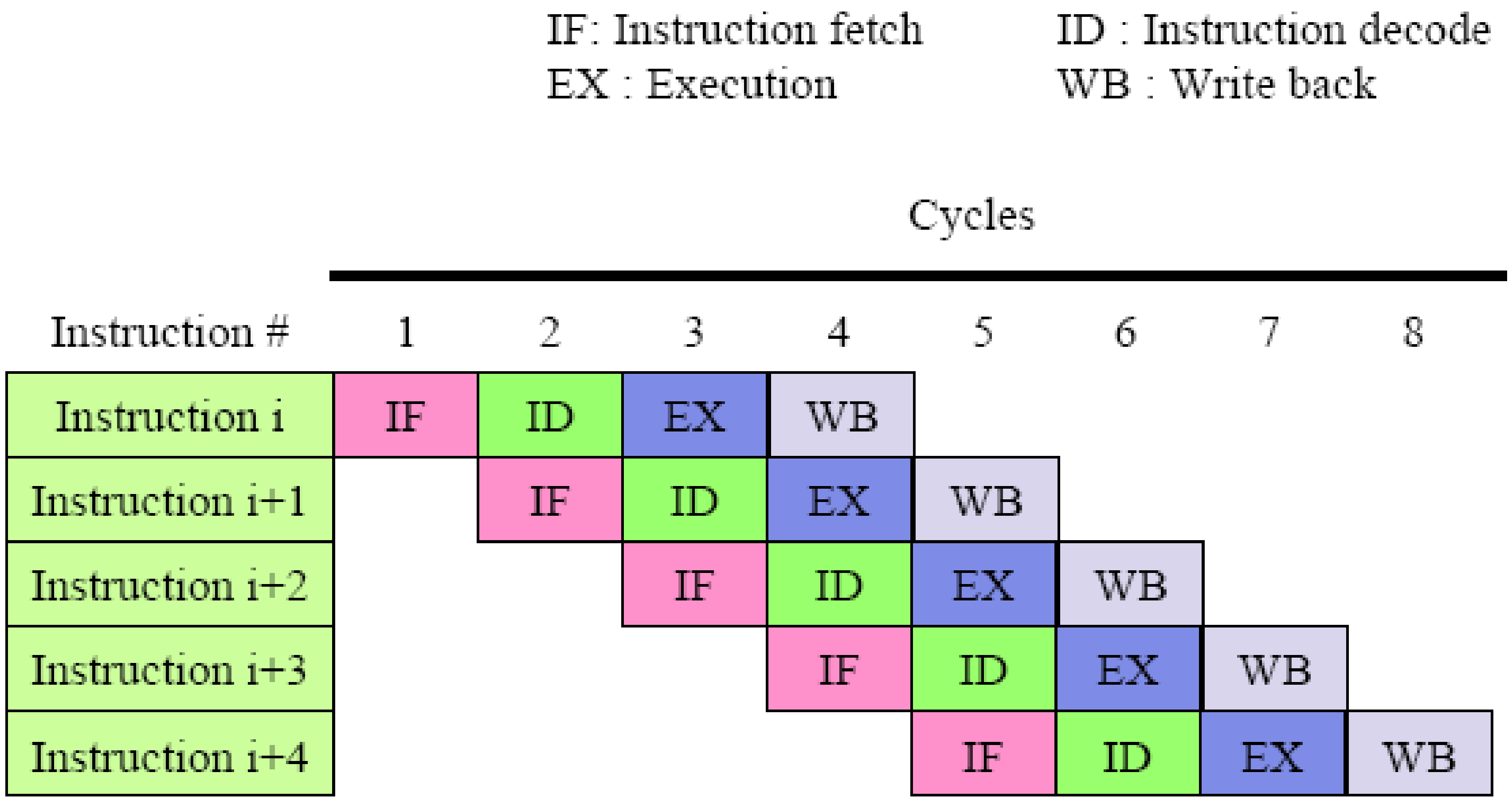
Sequential Execution	Instruction-Level Parallelism
1. a = 10 + 5 2. b = 12 + 7 3. c = a + b	1.A. a = 10 + 5 1.B. b = 12 + 7 2. c = a + b
Instructions: 3 Cycles: 3	Instructions: 3 Cycles: 2 (-33%)

Slide source: https://www.cs.wustl.edu/~jain/cse567-11/ftp/multcore.pdf

Exploring different level of parallelism

- **Instruction-level parallelism** (**ILP**): how many of the operations/instructions in a computer program can be performed simultaneously
  - 1. e = a + b
  - 2. f = c + d
  - 3. m = e * f
  - **1 and 2 can operate in parallel . 3 depends to 1 and 2.**
- *A* **superscalar processor** is a CPU that implements a form of parallelism called instruction-level parallelism within a single **processor**. It therefore allows faster CPU throughput (the number of instructions that can be executed in a unit of time) than would otherwise be possible at a given clock rate. Source: wiki

Pipelining Execution: Single CPU



IF: Instruction fetch
ID : Instruction decode
EX : Execution
WB : Write back

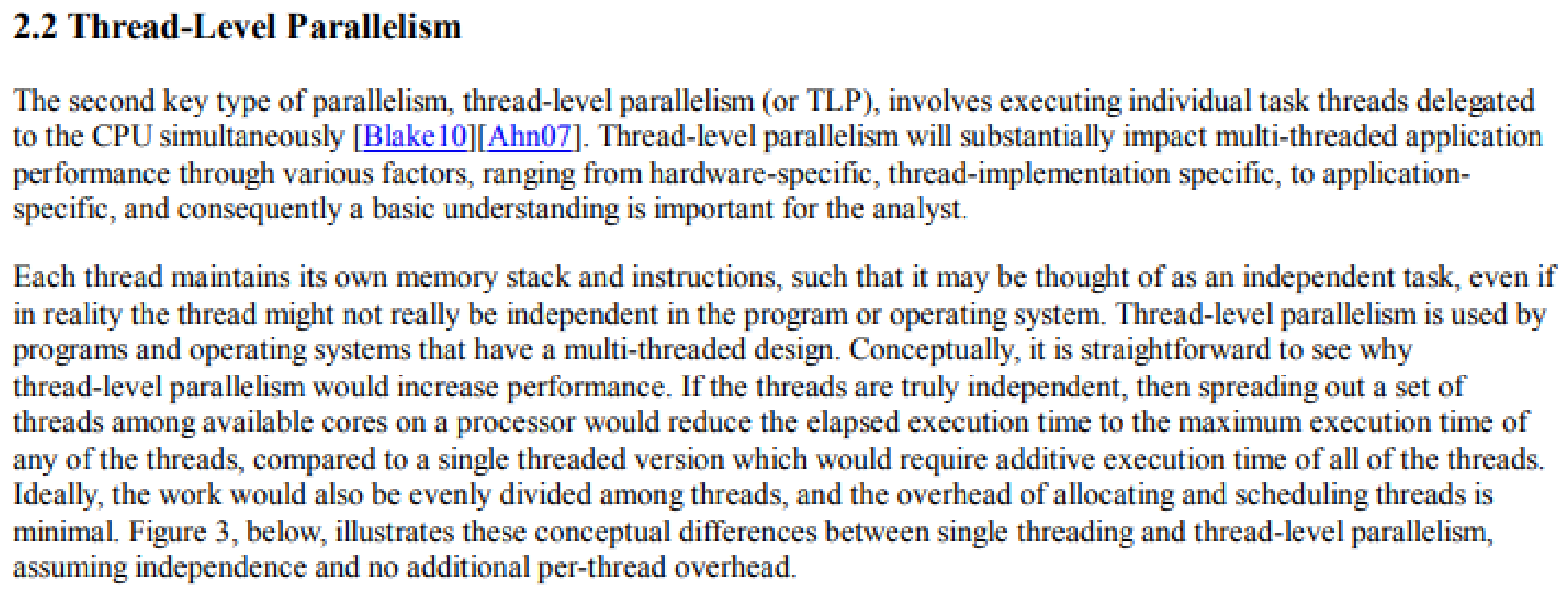
Cycles

Instruction #	1	2	3	4	5	6	7	8
Instruction i	IF	ID	EX	WB				
Instruction i+1		IF	ID	EX	WB			
Instruction i+2			IF	ID	EX	WB		
Instruction i+3				IF	ID	EX	WB	
Instruction i+4					IF	ID	EX	WB

Superscalar 2-issue pipleine

	Cycles						
Instruction type	1	2	3	4	5	6	7
Integer	IF	ID	EX	WB			
Floating point	IF	ID	EX	WB			
Integer		IF	ID	EX	WB		
Floating point		IF	ID	EX	WB		
Integer			IF	ID	EX	WB	
Floating point			IF	ID	EX	WB	
Integer				IF	ID	EX	WB
Floating point				IF	ID	EX	WB

2-issue super-scalar machine



2.2 Thread-Level Parallelism

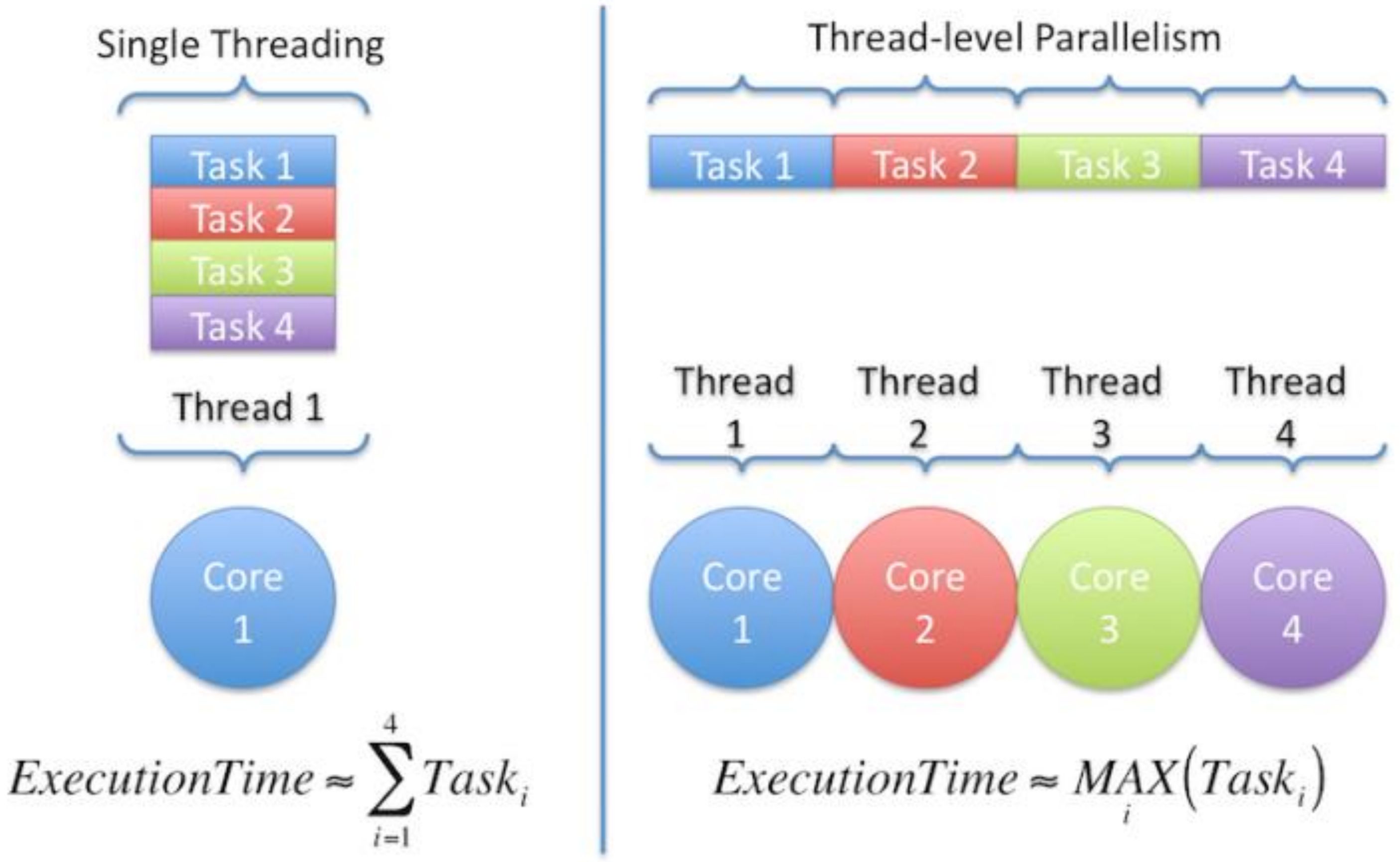
The second key type of parallelism, thread-level parallelism (or TLP), involves executing individual task threads delegated to the CPU simultaneously [Blake10][Ahn07]. Thread-level parallelism will substantially impact multi-threaded application performance through various factors, ranging from hardware-specific, thread-implementation specific, to application-specific, and consequently a basic understanding is important for the analyst.

Each thread maintains its own memory stack and instructions, such that it may be thought of as an independent task, even if in reality the thread might not really be independent in the program or operating system. Thread-level parallelism is used by programs and operating systems that have a multi-threaded design. Conceptually, it is straightforward to see why thread-level parallelism would increase performance. If the threads are truly independent, then spreading out a set of threads among available cores on a processor would reduce the elapsed execution time to the maximum execution time of any of the threads, compared to a single threaded version which would require additive execution time of all of the threads. Ideally, the work would also be evenly divided among threads, and the overhead of allocating and scheduling threads is minimal. Figure 3, below, illustrates these conceptual differences between single threading and thread-level parallelism, assuming independence and no additional per-thread overhead.

Slide source: https://www.cs.wustl.edu/~jain/cse567-11/ftp/multcore.pdf

Example for Thread Level Parallelism

This simplistic ideal model of thread-level parallelism performance is complicated by several other factors, such that the ideal scenario is rarely observed in real applications. Performance-impacting factors include the load balance, level of execution independence, thread-locking mechanisms, scheduling methods, and thread memory required. Further, data-level parallelism among the distributed threads may impact performance, as the subsequent section discusses. The thread implementation library in both the operating system and the specific application will also impact performance [Blake10] [Moseley07]. Consequently, an analyst examining an application with thread-level parallelism may need to control or regress these factors to quantify multi-core performance of the multi-threaded application.

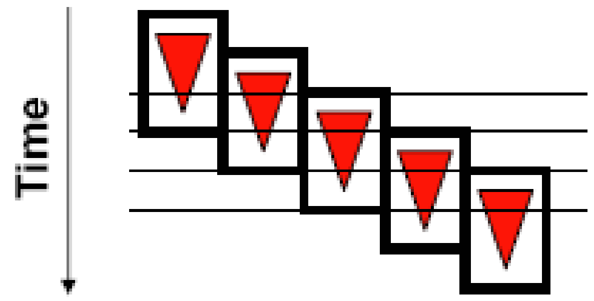


$$ExecutionTime \approx \sum_{i=1}^{4} Task_i$$

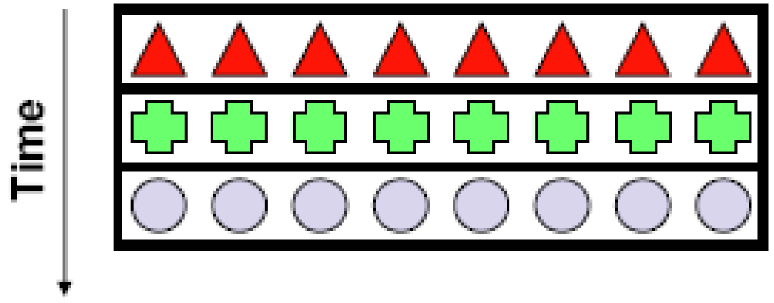
$$ExecutionTime \approx \underset{i}{MAX}\left(Task_i\right)$$

- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

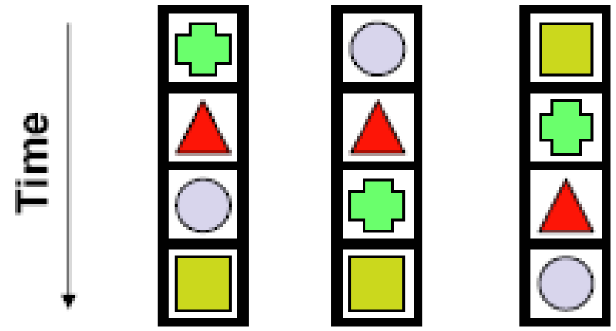
Types of Parallelism



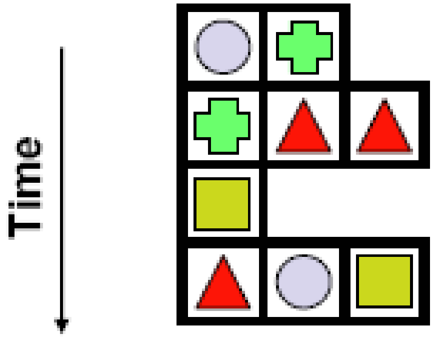
Pipelining



Data-Level Parallelism (DLP)



Thread-Level Parallelism (TLP)



Instruction-Level Parallelism (ILP)

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  1. Throughput of computers that run many programs
  2. Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

New Approach: Mulithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks
- When switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun’s Niagara

Source of Underutilization

- Stall or Underutilization:
  - L1 cache miss (Data and Instruction)
  - L2 cache miss
  - Instruction dependency
  - Long execution time operation, example: Divide
  - Branch miss prediction

Course-Grained Multithreading

- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn’t slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400

Simultaneous Multi-threading ...

One thread, 8 units

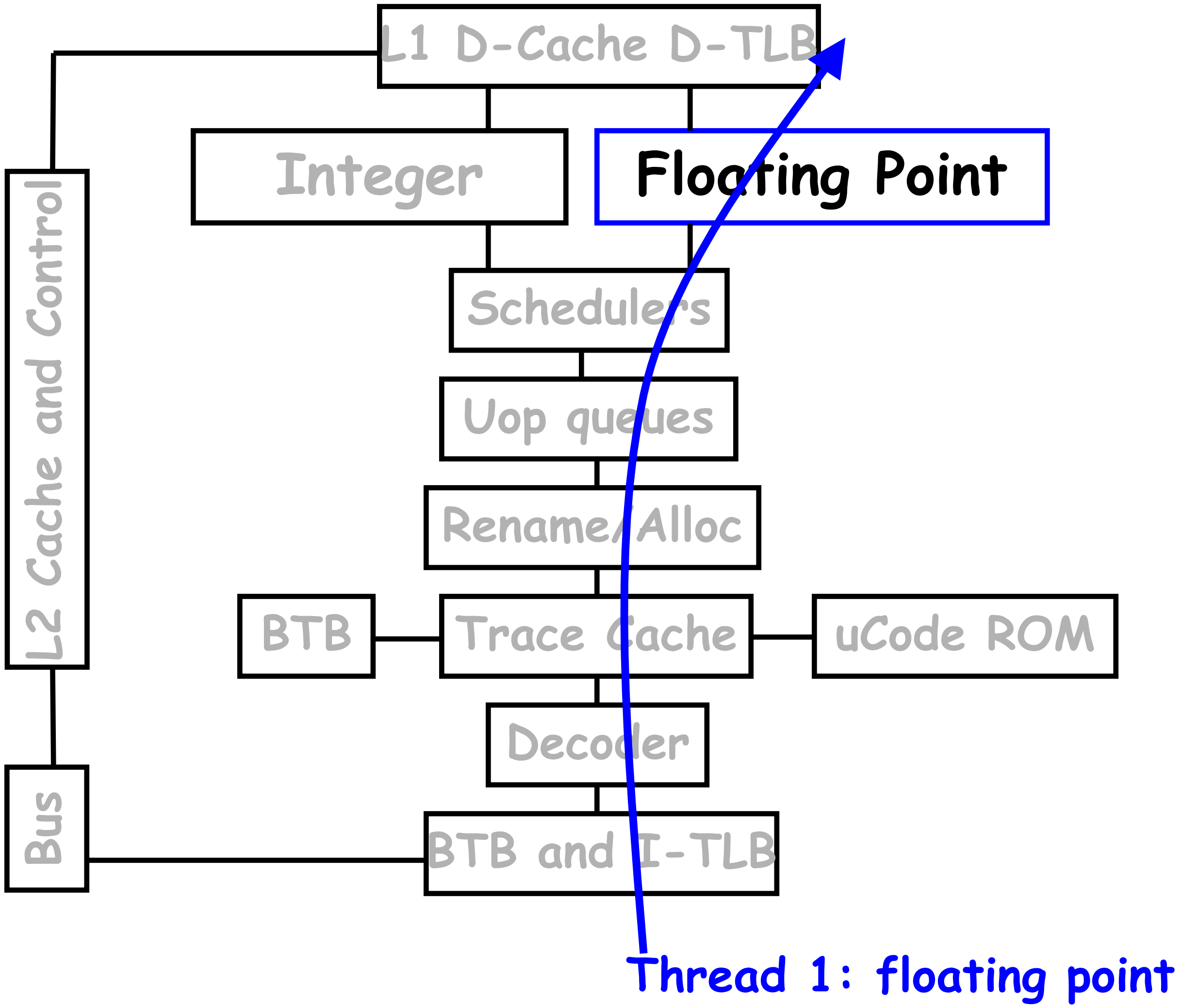
Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨							🟨
2	🟨	🟨					🟨	
3				🟨	🟨			
4								
5								
6								
7	🟨			🟨		🟨		
8		🟨			🟨			
9				🟨				

Two threads, 8 units

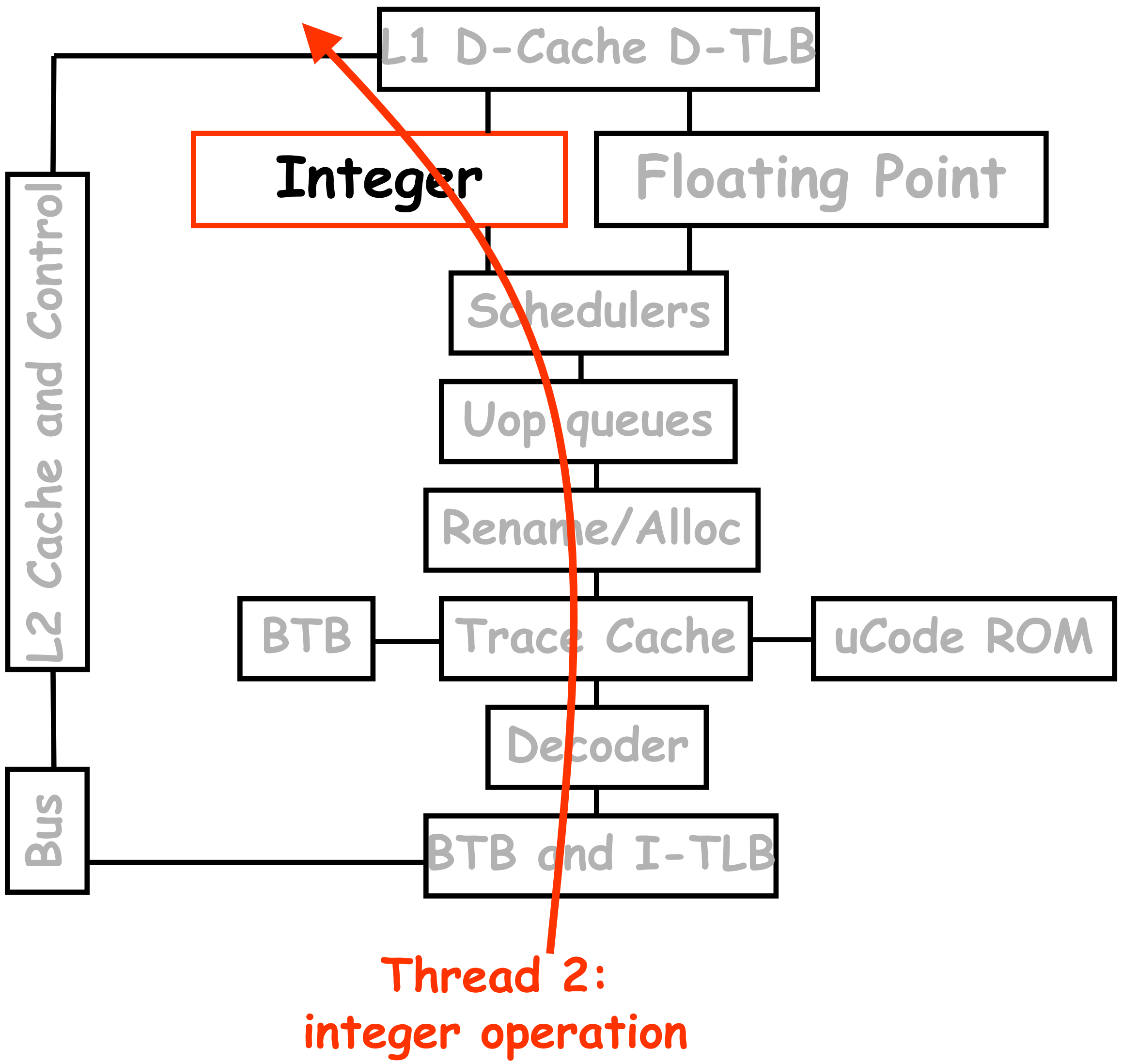
Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨	🟦	🟦					🟨
2	🟨	🟨	🟦			🟦	🟨	
3	🟦			🟨	🟨			
4	🟦	🟦				🟦		
5		🟦						🟦
6								
7	🟨		🟦	🟨	🟦	🟨		
8		🟨		🟦	🟨	🟦		
9	🟦	🟦		🟨		🟦		

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

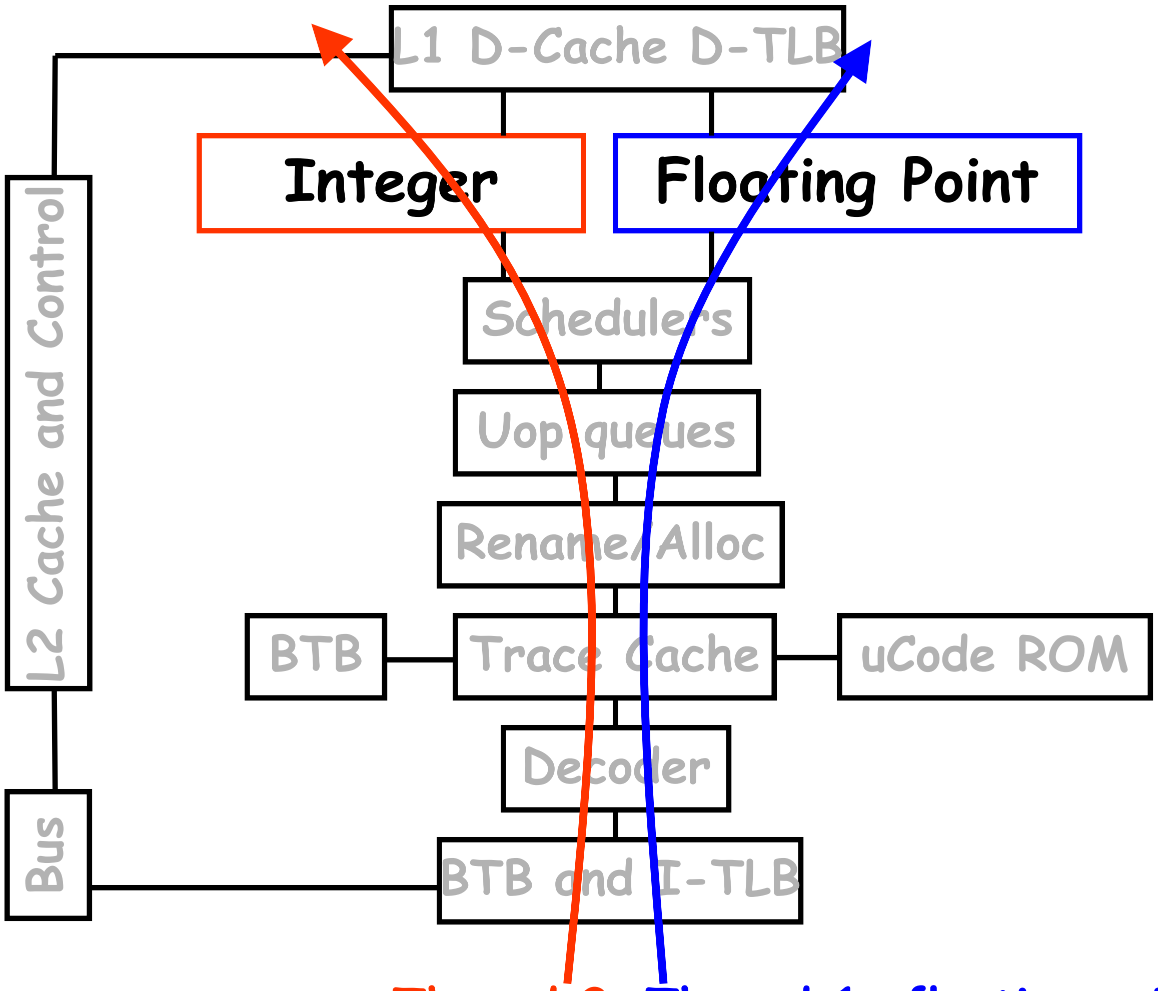
Without SMT, only a single thread can run at any given time



Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



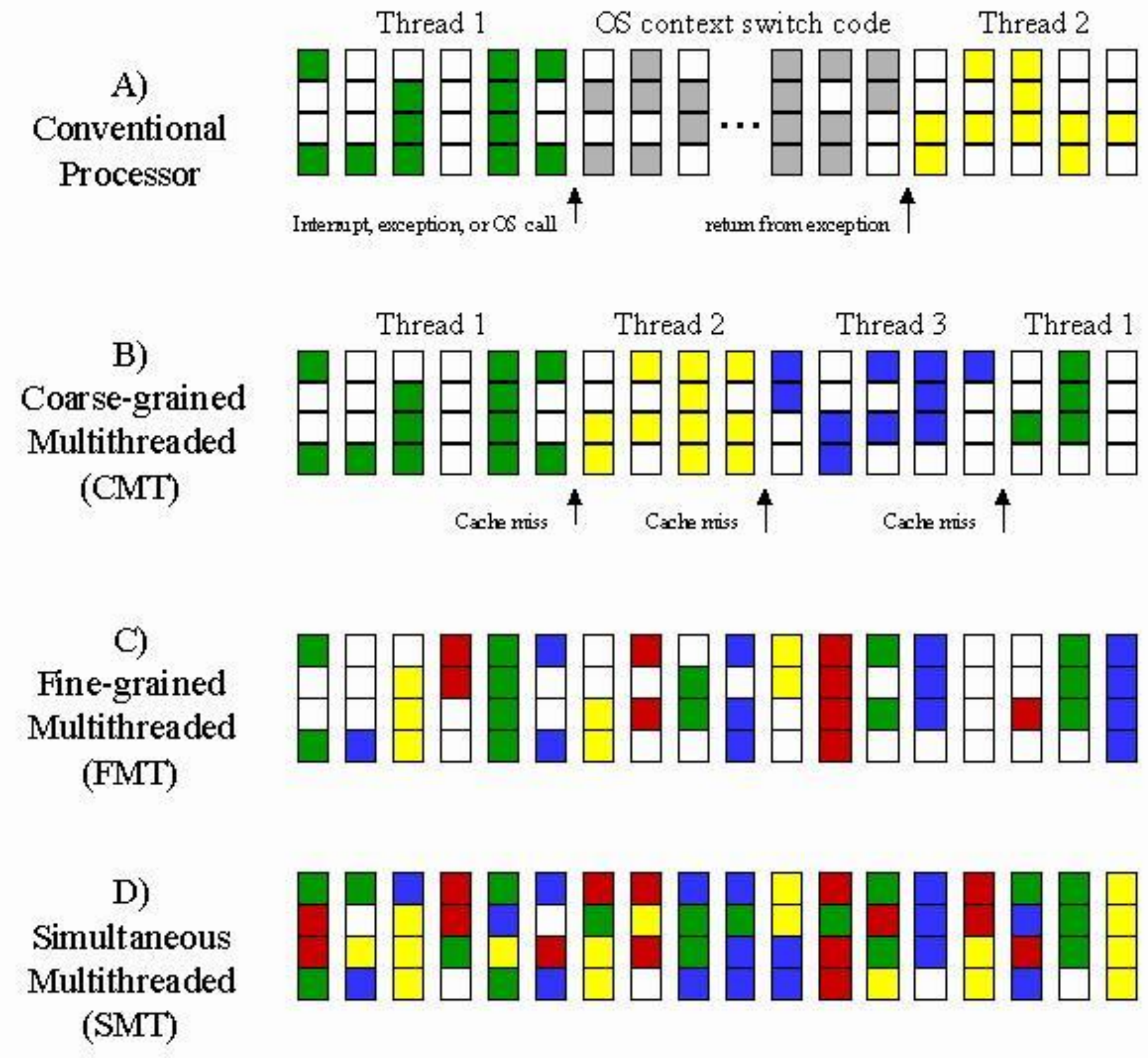
Thread 2: integer operation

Thread 1: floating point

Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

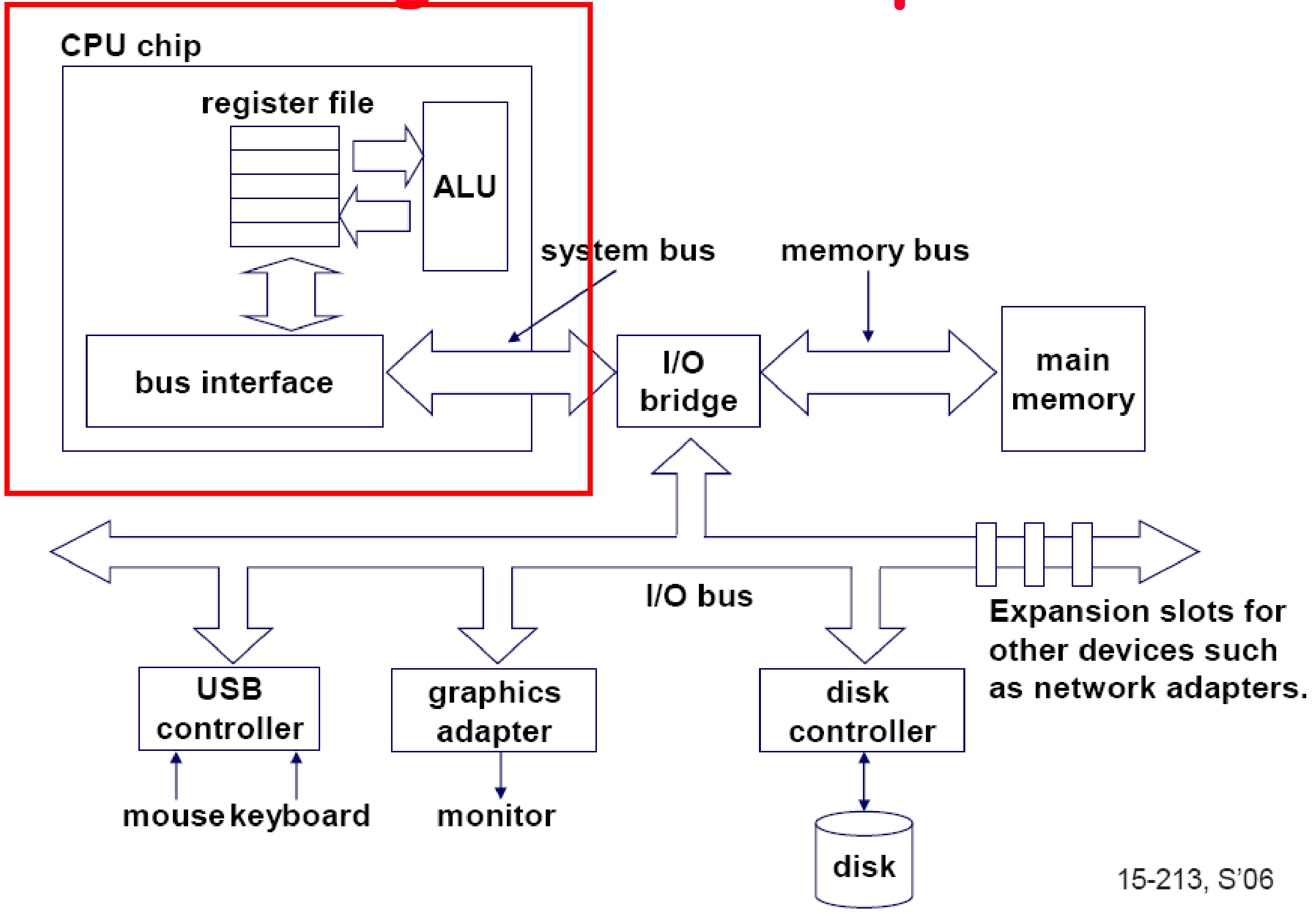
Source: Micrprocessor Report, December 6, 1999
"Compaq Chooses SMT for Alpha"


Thread 1
OS context switch code
Thread 2
A) Conventional Processor
Interrupt, exception, or OS call
return from exception
Thread 1
Thread 2
Thread 3
Thread 1
B) Coarse-grained Multithreaded (CMT)
Cache miss
Cache miss
Cache miss
C) Fine-grained Multithreaded (FMT)
D) Simultaneous Multithreaded (SMT)

- each color corresponds to a thread, example: Green, Yellow, Blue, Red all are threads.
- **Important thing to watch in below only in SMT we are able to execute more than one thread at a time.**

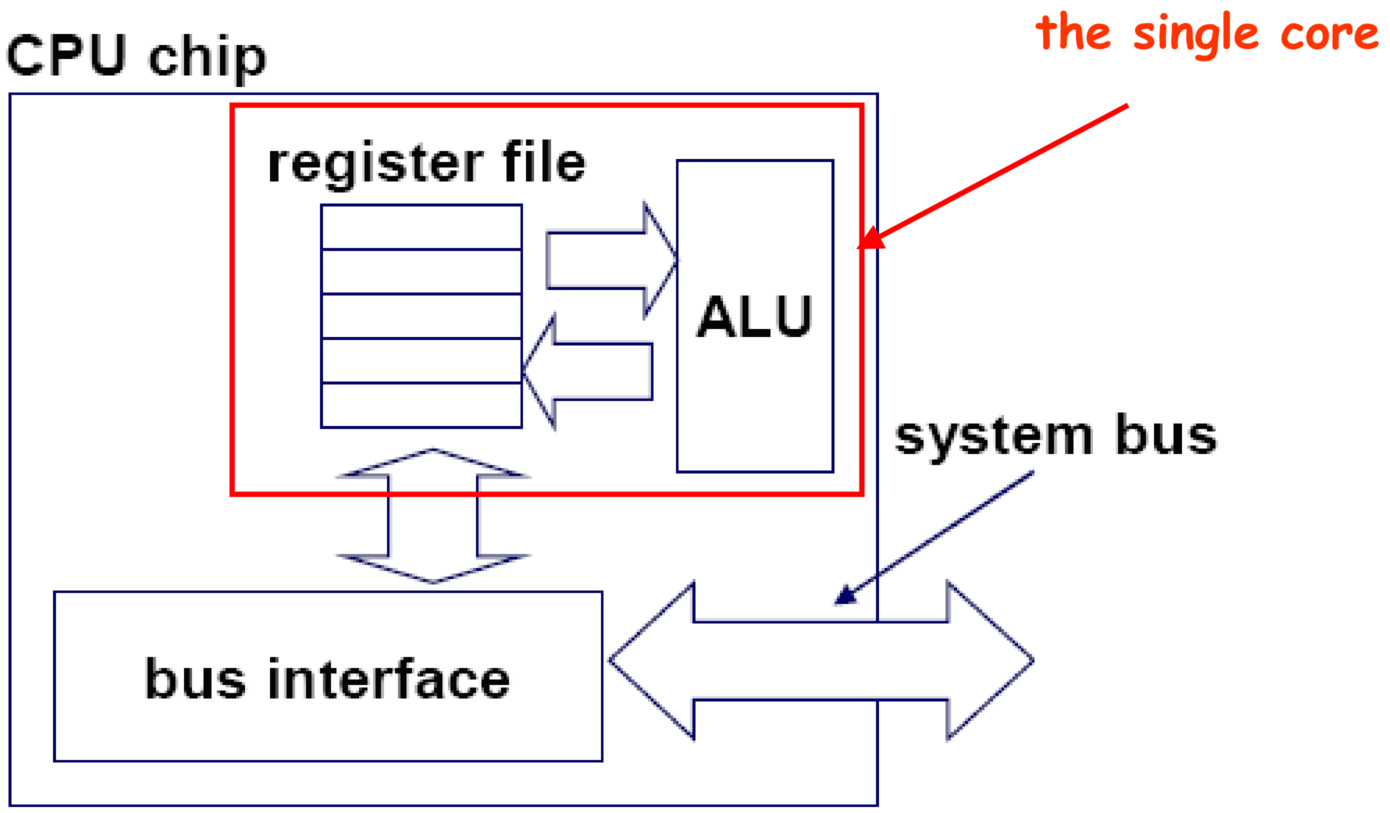
Multi-core architectures

Single-core computer



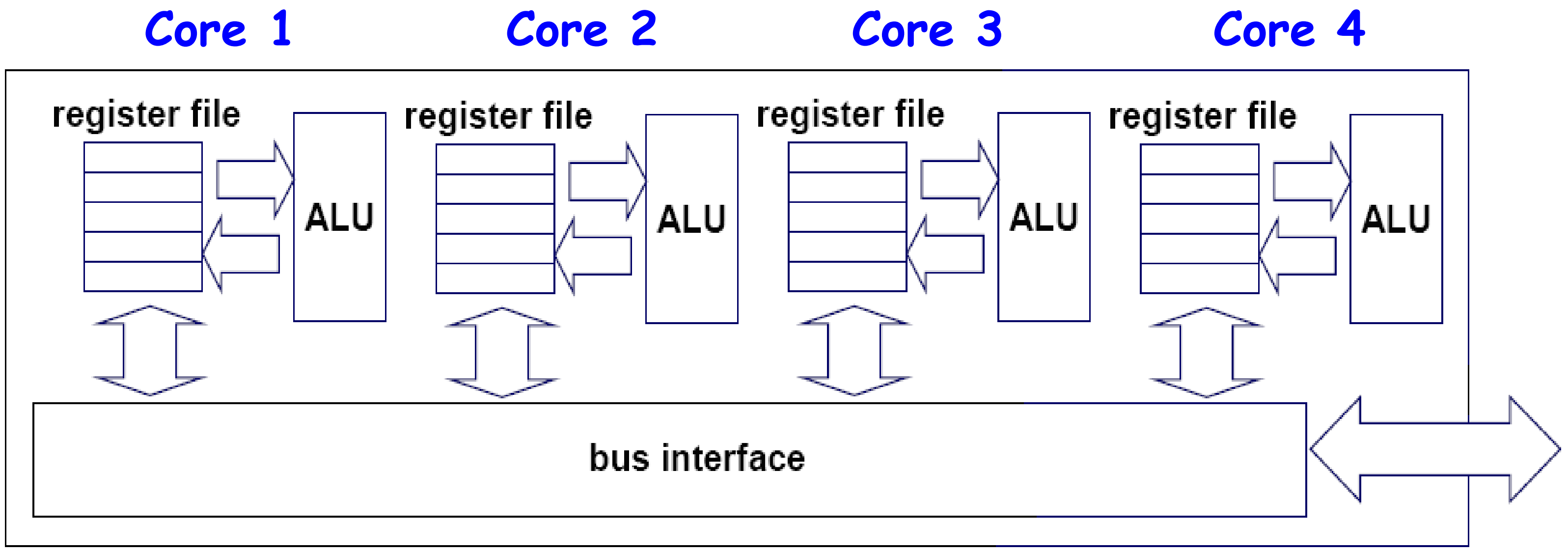
15-213, S'06

Single-core CPU chip



Multi-core architectures

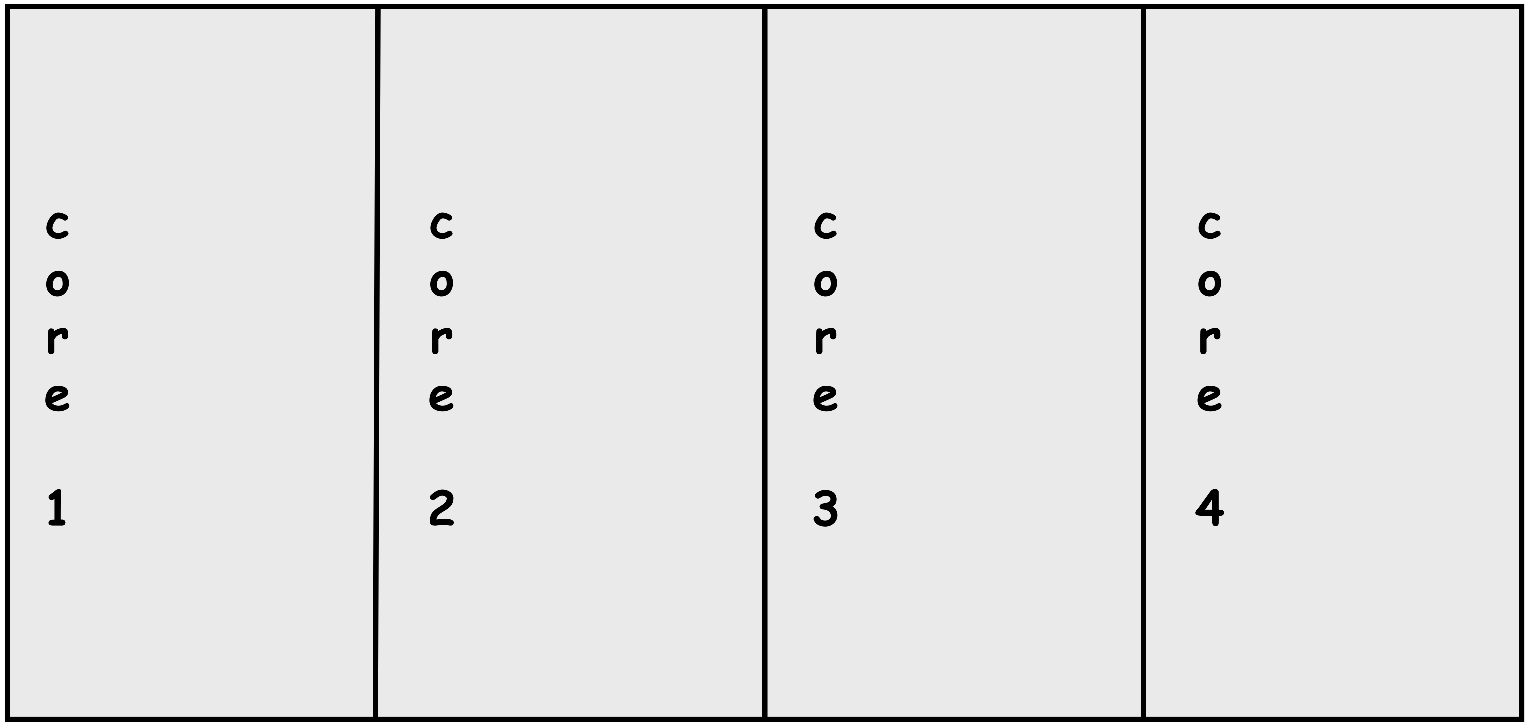
- New trend in computer architecture:
  Replicate multiple processor cores on a single die.



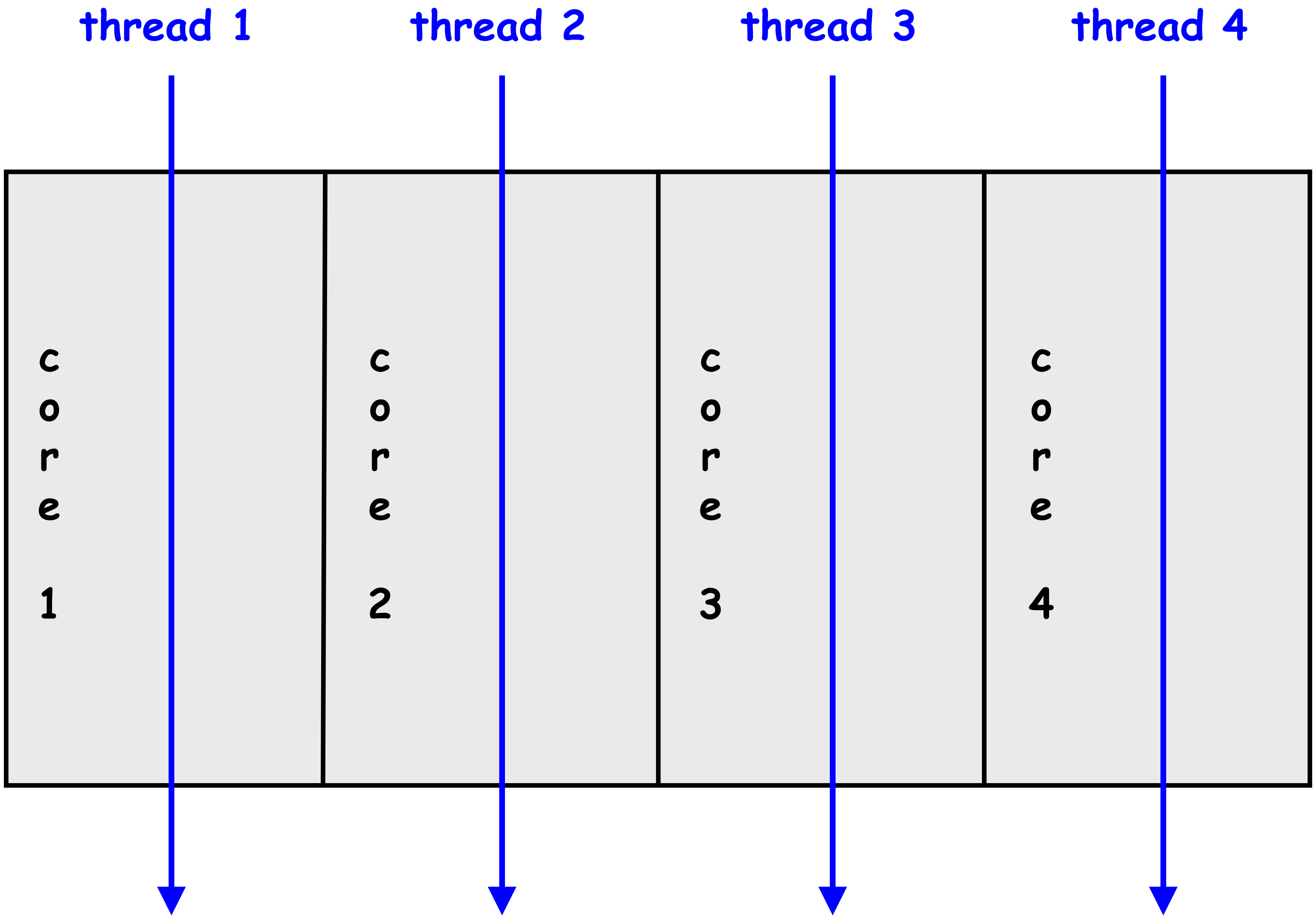
Multi-core CPU chip

Multi-core CPU chip

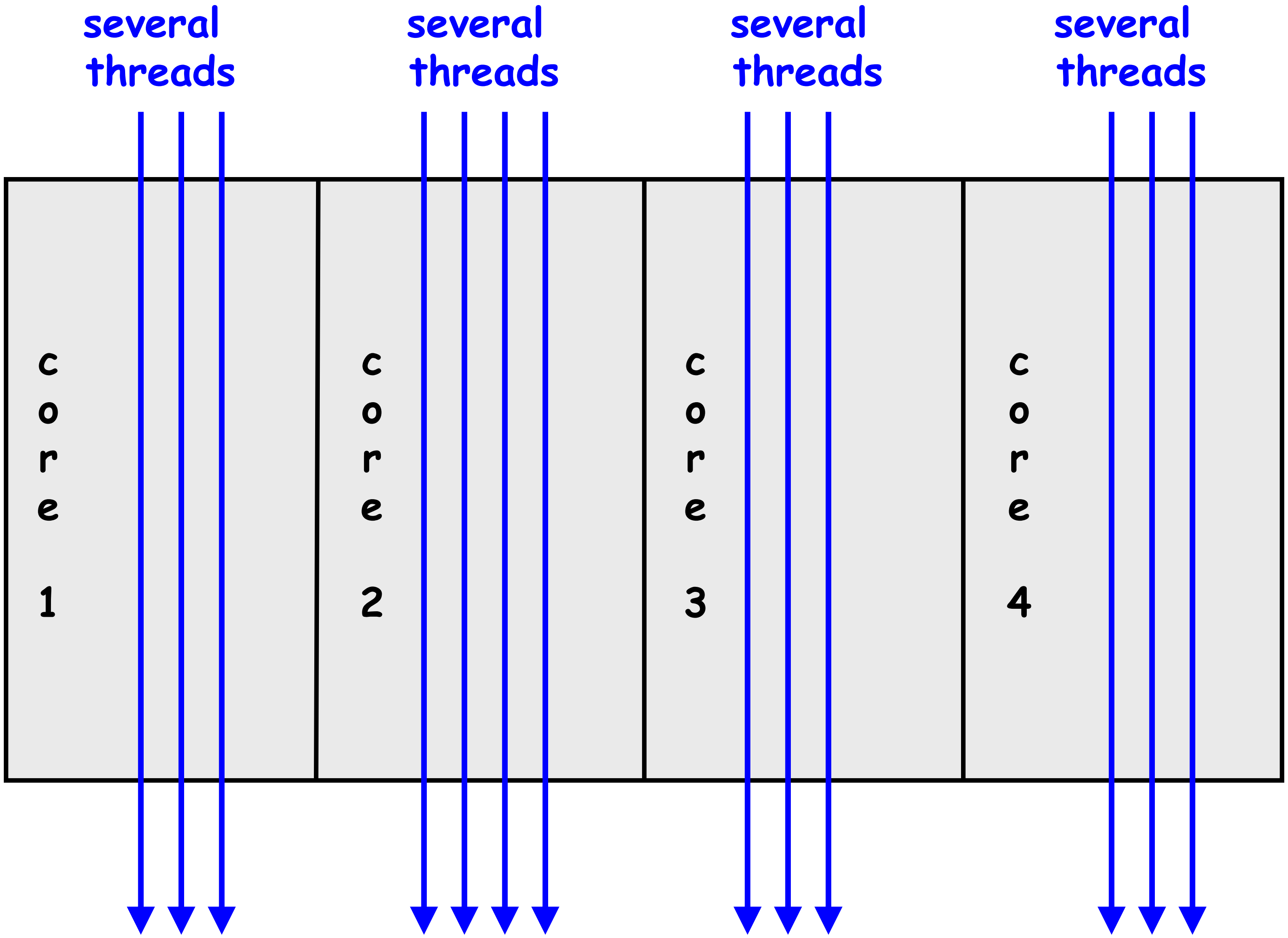
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

General context: Multiprocessors

- Multiprocessor is any computer with several processors
- SIMD
  - Single instruction, multiple data
  - Modern graphics cards
- MIMD
  - Multiple instructions, multiple data

Lemieux cluster, Pittsburgh supercomputing center

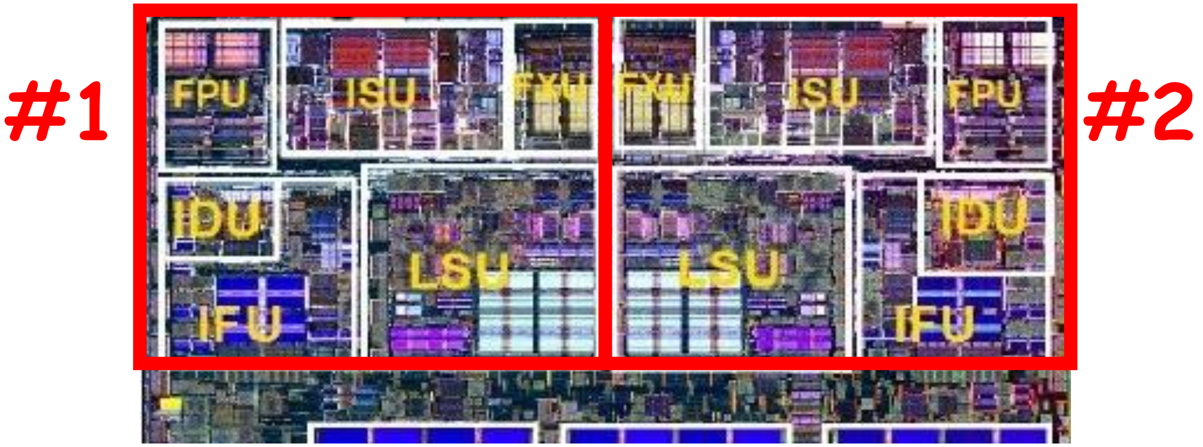
Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

As programmers, do we care?

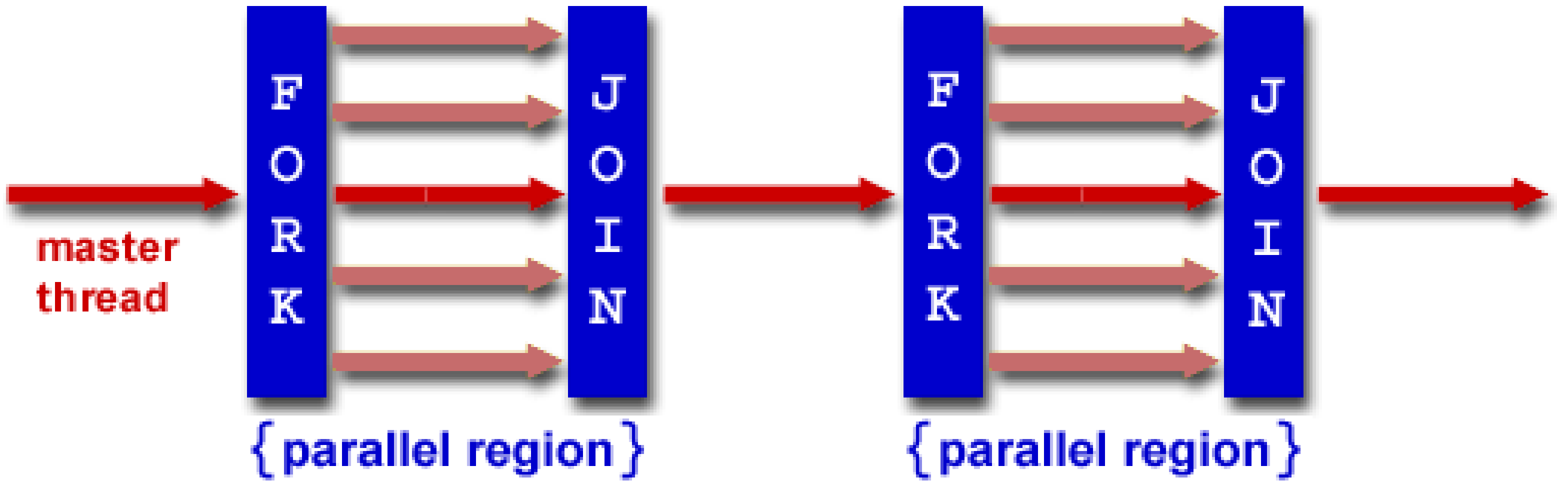
- What happens if we run a program on a multi-core?

```
void
array_add(int A[], int B[], int C[], int length)
  {
  int i;
  for (i = 0 ; i < length ; ++i) {
   C[i] = A[i] + B[i];
  }
}
```



What if we want a program to run on both processors?

- We have to explicitly tell the machine exactly how to do this
  - This is called parallel programming or concurrent programming
- There are many parallel/concurrent programming models
  - We will look at a relatively simple one: fork-join parallelism

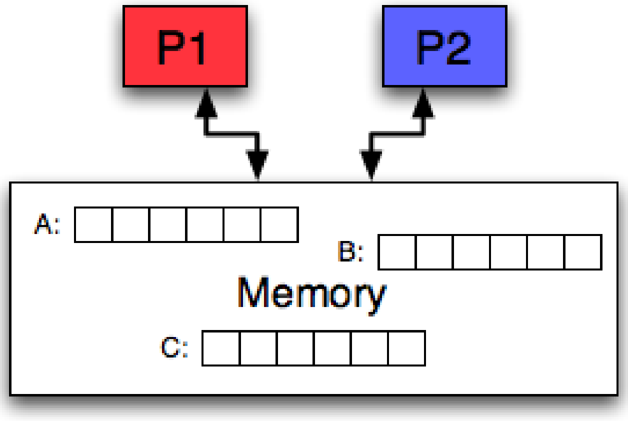


Fork/Join Logical Example

1. Fork N-1 threads
2. Break work into N pieces (and do it)
3. Join (N-1) threads

```
void
array_add(int A[], int B[], int C[], int length) {
   cpu_num = fork(N-1);
  int i;
  for (i = cpu_num ; i < length ; i += N) {
    C[i] = A[i] + B[i];
  }
   join();
}
```

How good is this with caches?

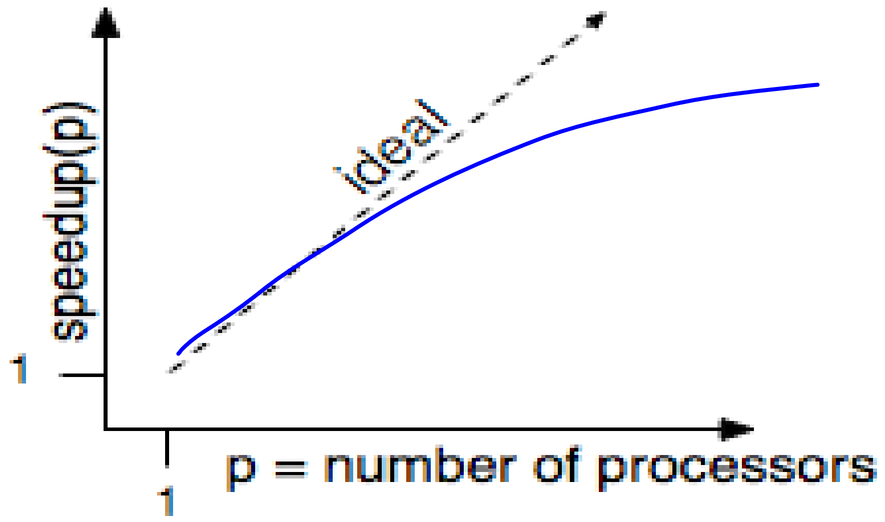


How does this help performance?

- Parallel speedup measures improvement from parallelization:

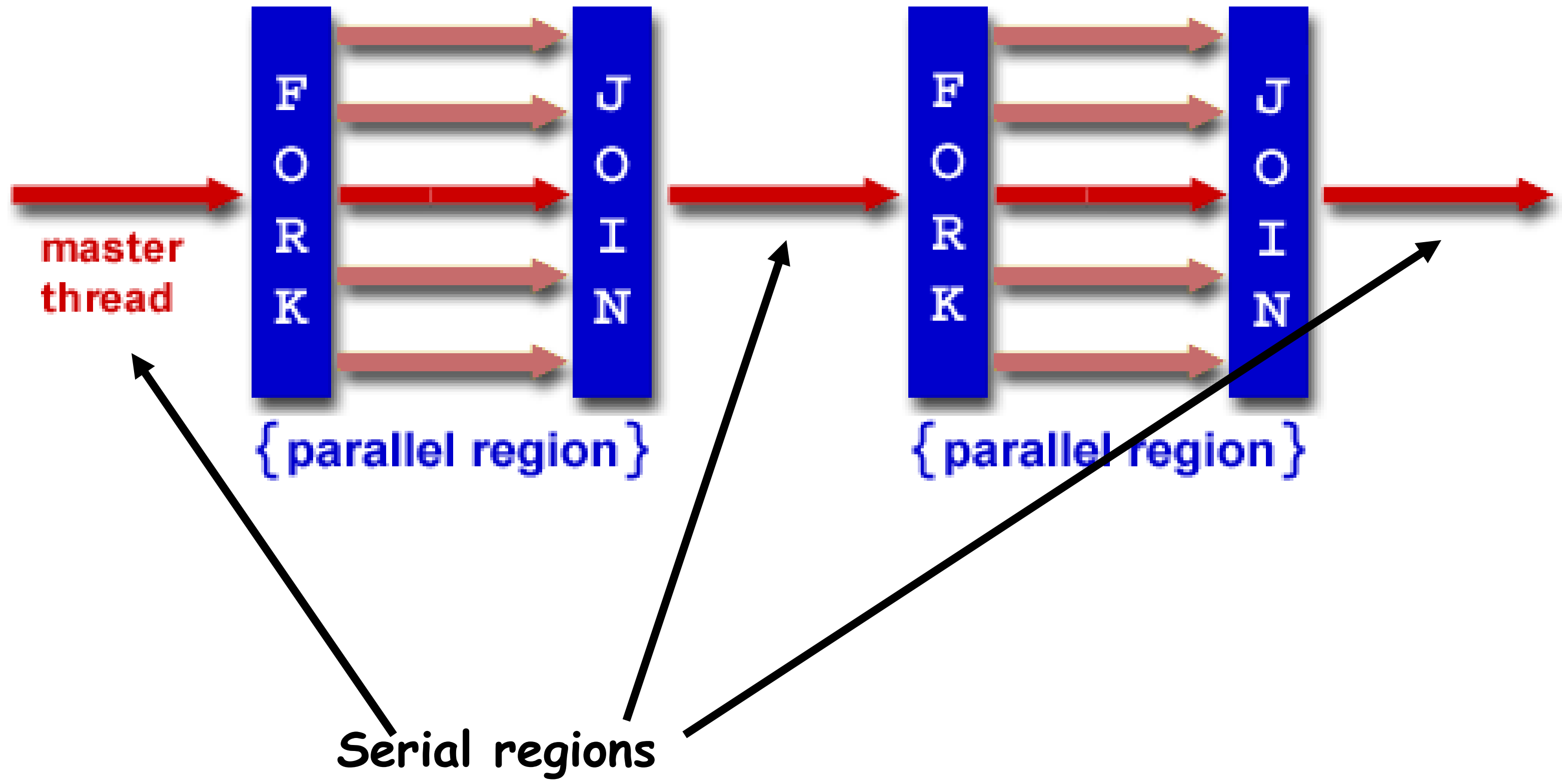
$$\text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with p processors}}$$

- What can we realistically expect?



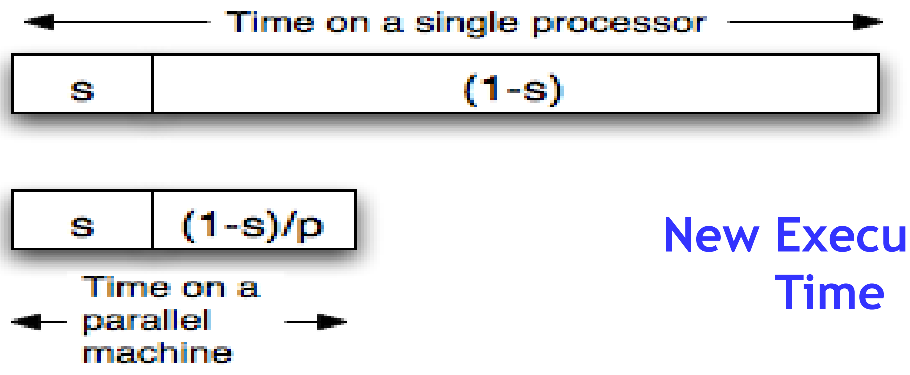
Reason #1: Amdahl's Law

In general, the whole computation is not (easily) parallelizable



Reason #1: Amdahl's Law

- Suppose a program takes 1 unit of time to execute serially
- A fraction of the program, s, is inherently serial (unparallelizable)



New Execution Time $= \frac{1-s}{P} + s$

- For example, consider a program that, when executing on one processor, spends 10% of its time in a non-parallelizable region. How much faster will this program run on a 3-processor system?

New Execution Time $= \frac{.9T}{3} + .1T =$ Speedup $=$

- What is the maximum speedup from parallelization?

Reason #2: Overhead

```
void
array_add(int A[], int B[], int C[], int length) {
    cpu_num = fork(N-1);
   int i;
   for (i = cpu_num ; i < length ; i += N) {
     C[i] = A[i] + B[i];
   }
    join();
}
```

- Forking and joining is not instantaneous
  - Involves communicating between processors
  - May involve calls into the operating system
    - Depends on the implementation

$$\text{New Execution Time} = \frac{1-s}{P} + s + \text{overhead}(P)$$