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., $y = \text{sign}(a), a = a*2$;
  or
  - Only one access at a time: *mutual exclusion*
Safety: Example

- “The *too much milk* problem”

<table>
<thead>
<tr>
<th>time</th>
<th>You</th>
<th>Your Roommate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Arrive home</td>
<td>Arrive home</td>
</tr>
<tr>
<td>3:05</td>
<td>Look in fridge, no milk</td>
<td>Look in fridge, no milk</td>
</tr>
<tr>
<td>3:10</td>
<td>Leave for grocery</td>
<td>Leave for grocery</td>
</tr>
<tr>
<td>3:15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive at grocery</td>
<td></td>
</tr>
<tr>
<td>3:25</td>
<td>Buy milk</td>
<td>Buy Milk</td>
</tr>
<tr>
<td>3:35</td>
<td>Arrive home, put milk in fridge</td>
<td>Arrive home, put up milk</td>
</tr>
<tr>
<td>3:45</td>
<td></td>
<td>Oh no!</td>
</tr>
<tr>
<td>3:50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

*Does this work?* **too much milk**
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

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

synchronized void post(int v) {
    balance = balance + v;
}
```
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. $m = e \times f$
  • 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

<table>
<thead>
<tr>
<th>Instruction #</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction i</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Superscalar 2-issue pipeline**

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating point</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**2-issue super-scalar machine**
Types of parallelism

- Pipelining, Instruction Level parallelism (ILP), Thread Level Parallelism (TLP) and Data Level Parallelism (DLP)

- **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: Multithreaded 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 $\approx 100$s to $1000$s 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

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two threads, 8 units

<table>
<thead>
<tr>
<th>Cycle</th>
<th>M</th>
<th>M</th>
<th>FX</th>
<th>FX</th>
<th>FP</th>
<th>FP</th>
<th>BR</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Thread 1: floating point
Without SMT, only a single thread can run at any given time.
SMT processor: both threads can run concurrently
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”
Multithreaded Categories

- **Superscalar**
- **Fine-Grained**
- **Coarse-Grained**
- **Multiprocessing**
- **Simultaneous Multithreading**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
<th>Thread 4</th>
<th>Thread 5</th>
<th>Idle slot</th>
</tr>
</thead>
</table>

- **LD**
- **FX**
- **FP**
- **BR**
Multi-core architectures
Single-core computer

- CPU chip
  - register file
  - ALU
- bus interface
- system bus
- memory bus
- I/O bridge
- main memory
- I/O bus
- Expansion slots for other devices such as network adapters.

- USB controller
  - mouse keyboard
- graphics adapter
  - monitor
- disk controller
  - disk
Single-core CPU chip

CPU chip

register file

ALU

bus interface

system bus

the single core
Multi-core architectures

New trend in computer architecture: Replicate multiple processor cores on a single die.
## Multi-core CPU chip

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

<table>
<thead>
<tr>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
The cores run in parallel

thread 1  thread 2  thread 3  thread 4

core 1  core 2  core 3  core 4
Within each core, threads are time-sliced (just like on a uniprocessor)

- Several threads
- Several threads
- Several threads
- Several threads

<table>
<thead>
<tr>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
<th>Core 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Threads</td>
<td>Threads</td>
<td>Threads</td>
</tr>
</tbody>
</table>
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
- ILP is limited
- 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?

```c
void array_add(int A[], int B[], int C[], int length) {
    int i;
    for (i = 1 ; i < 11 ; ++i) {
        C[i] = A[i] + B[i];
    }
}
```

#1 #2
## Processor Clk speed trends

<table>
<thead>
<tr>
<th>Year</th>
<th>CPU</th>
<th>MHz</th>
<th>Transistors</th>
<th>Process</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1978</td>
<td>8086</td>
<td>4.77</td>
<td>29K</td>
<td>3000nm</td>
<td>2</td>
</tr>
<tr>
<td>1982</td>
<td>80286</td>
<td>8</td>
<td>134K</td>
<td>1500nm</td>
<td>3</td>
</tr>
<tr>
<td>1985</td>
<td>80386</td>
<td>16</td>
<td>275K</td>
<td>1000nm</td>
<td>3</td>
</tr>
<tr>
<td>1989</td>
<td>80486</td>
<td>25</td>
<td>1.2M</td>
<td>1000nm</td>
<td>5</td>
</tr>
<tr>
<td>1993</td>
<td>Pentium</td>
<td>60</td>
<td>3.1M</td>
<td>800nm</td>
<td>5</td>
</tr>
<tr>
<td>1995</td>
<td>Pentium Pro</td>
<td>150</td>
<td>7.5M</td>
<td>600nm</td>
<td>12</td>
</tr>
<tr>
<td>1997</td>
<td>Pentium II</td>
<td>233</td>
<td>7.5M</td>
<td>350nm</td>
<td>12</td>
</tr>
<tr>
<td>1999</td>
<td>Pentium III</td>
<td>450</td>
<td>9.5M</td>
<td>250nm</td>
<td>12</td>
</tr>
<tr>
<td>2000</td>
<td>Willamette P4</td>
<td>1400</td>
<td>42M</td>
<td>180nm</td>
<td>20</td>
</tr>
<tr>
<td>2002</td>
<td>Northwood P4</td>
<td>2000</td>
<td>55M</td>
<td>130nm</td>
<td>20</td>
</tr>
<tr>
<td>2004</td>
<td>Prescott P4</td>
<td>3400</td>
<td>125M</td>
<td>90nm</td>
<td>31</td>
</tr>
<tr>
<td>2001</td>
<td>Itanium</td>
<td>733</td>
<td>25M</td>
<td>180nm</td>
<td>10</td>
</tr>
<tr>
<td>2002</td>
<td>Itanium 2</td>
<td>1000</td>
<td>220M</td>
<td>180nm</td>
<td>8</td>
</tr>
</tbody>
</table>
### Processor Clk speed trends

<table>
<thead>
<tr>
<th>Brand Name &amp; Processor Number</th>
<th>Base Clock Speed</th>
<th>Turbo Frequency</th>
<th>Cores/Threads</th>
<th>Cache</th>
<th>Memory Speed Support DDR-3</th>
<th>TDP</th>
<th>Package</th>
<th>Pricing (1Ku)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Core™ i7-620M</td>
<td>2.66 GHz</td>
<td>Up to 3.33 GHz</td>
<td>2/4</td>
<td>4 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>$332</td>
</tr>
<tr>
<td>Intel® Core™ i7-640LM</td>
<td>2.13 GHz</td>
<td>Up to 2.93 GHz</td>
<td>2/4</td>
<td>4 MB</td>
<td>1066</td>
<td>25 W</td>
<td>BGA</td>
<td>$332</td>
</tr>
<tr>
<td>Intel® Core™ i7-620LM</td>
<td>2.0 GHz</td>
<td>Up to 2.8 GHz</td>
<td>2/4</td>
<td>4 MB</td>
<td>1066</td>
<td>25 W</td>
<td>BGA</td>
<td>$300</td>
</tr>
<tr>
<td>Intel® Core™ i7-640UM</td>
<td>1.20 GHz</td>
<td>Up to 2.26 GHz</td>
<td>2/4</td>
<td>4 MB</td>
<td>800</td>
<td>18 W</td>
<td>BGA</td>
<td>$305</td>
</tr>
<tr>
<td>Intel® Core™ i7-620UM</td>
<td>1.06 GHz</td>
<td>Up to 2.13 GHz</td>
<td>2/4</td>
<td>4 MB</td>
<td>800</td>
<td>18 W</td>
<td>BGA</td>
<td>$278</td>
</tr>
<tr>
<td>Intel® Core™ i5-540M</td>
<td>2.53 GHz</td>
<td>Up to 3.06 GHz</td>
<td>2/4</td>
<td>3 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>$257</td>
</tr>
<tr>
<td>Intel® Core™ i5-520M</td>
<td>2.40 GHz</td>
<td>Up to 2.93 GHz</td>
<td>2/4</td>
<td>3 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>$225</td>
</tr>
<tr>
<td>Intel® Core™ i5-520UM</td>
<td>1.06 GHz</td>
<td>Up to 1.86 GHz</td>
<td>2/4</td>
<td>3 MB</td>
<td>800</td>
<td>18 W</td>
<td>BGA</td>
<td>$241</td>
</tr>
<tr>
<td>Intel® Core™ i5-430M</td>
<td>2.26 GHz</td>
<td>Up to 2.53 GHz</td>
<td>2/4</td>
<td>3 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>Not listed</td>
</tr>
<tr>
<td>Intel® Core™ i3-350M</td>
<td>2.26 GHz</td>
<td>N/A</td>
<td>2/4</td>
<td>3 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>Not listed</td>
</tr>
<tr>
<td>Intel® Core™ i3-330M</td>
<td>2.13 GHz</td>
<td>N/A</td>
<td>2/4</td>
<td>3 MB</td>
<td>1066</td>
<td>35 W</td>
<td>rPGA BGA</td>
<td>Not listed</td>
</tr>
</tbody>
</table>
Multicore trends
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

```c
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:

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

♦ What can we realistically expect?
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)

\[
\text{New Execution Time} = \frac{1-s}{p} + s
\]

- 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 = \text{Speedup} =

- What is the maximum speedup from parallelization?
Reason #2: Overhead

```c
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

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