

AMATH 483/583
High Performance Scientific
Computing

Lecture 10:

Processes, Threads, Concurrency, Parallelism

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

Announcements

- Mid Term out this noon 04/28/2022 due 11:59AM 05/05/2022
- **The exam may not be discussed with anyone except course instructors**
- You may contact the instructors via *private* messages on Piazza to clarify questions

Overview

- Review
 - SISD, SIMD
- Multiple cores (MIMD)
- Concurrency
- Processes
- Parallelism
- Threads
 - Multithreading

Supercomputers (HPC)



Schematically

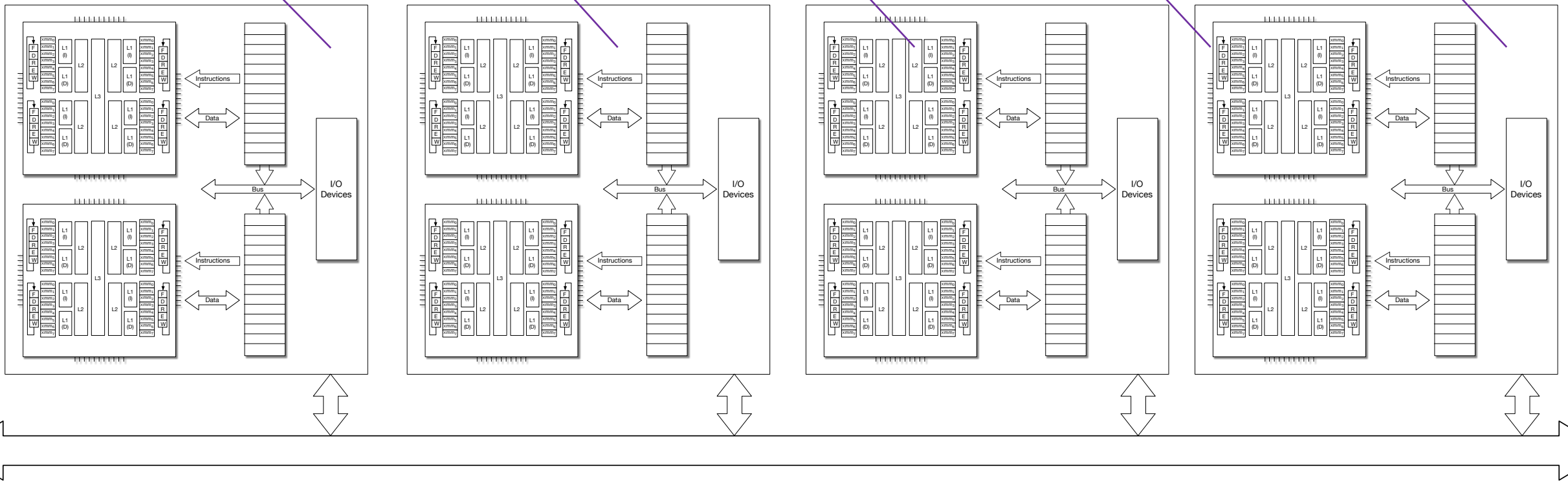
Put sockets on a blade

Put blades in a chassis

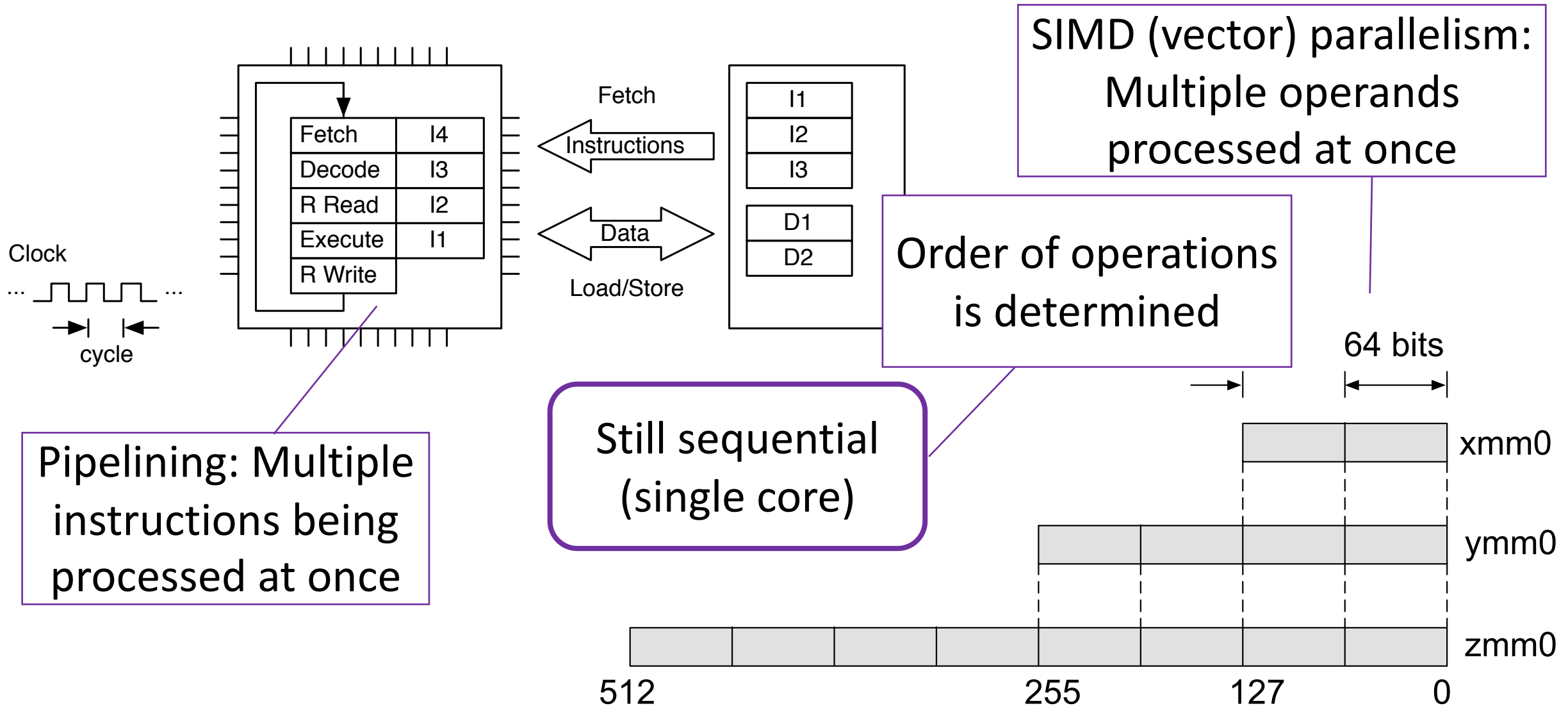
Put chassis in a rack

Put racks in a center

Put centers in the cloud



Parallelism and HPC so far



General Performance Principles

- Work harder

- Faster core

- Work smarter

- Branch predictions, etc.
- Better compilation
- Better algorithm
- Better implementation

- More workers

- More cores

Dennard scaling (ended 2005)

Higher optimization level, e.g.,
-O2, -O3, -Ofast, etc.

e.g., Strassen's algorithm

We did this

Parallel Computing

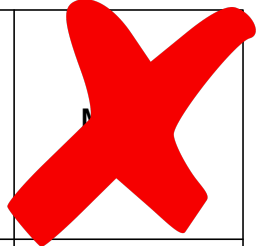
Flynn's Taxonomy (Aside)

Anyone in HPC must know Flynn's taxonomy

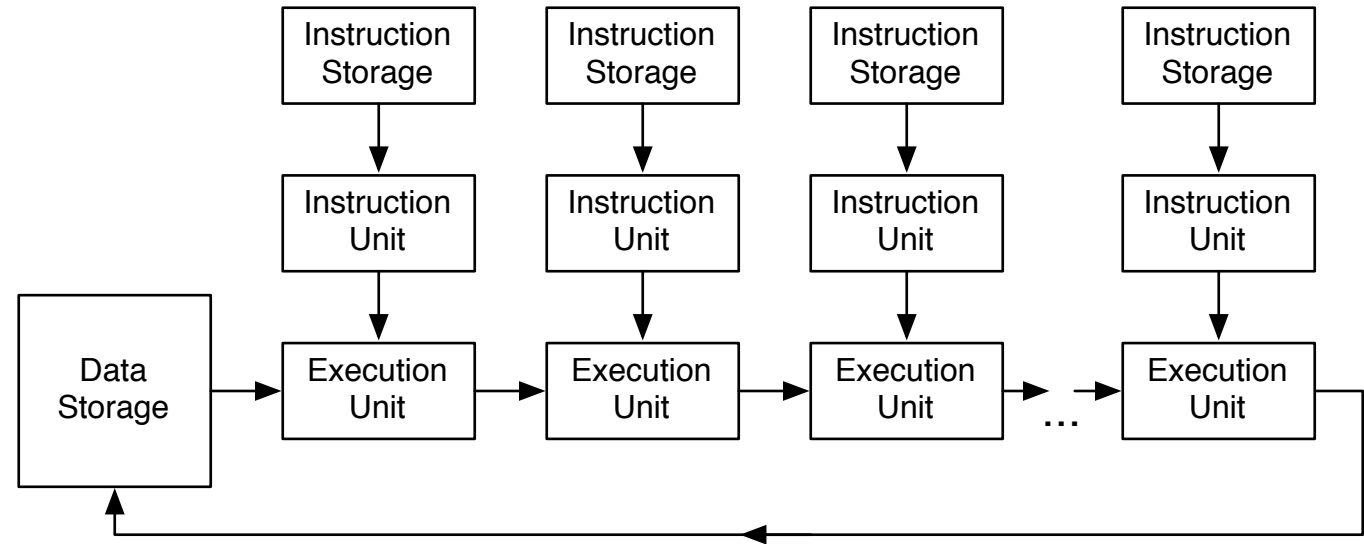
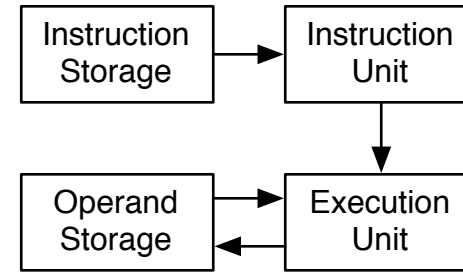
- **Classic** classification of parallel architectures (Michael Flynn, 1966)

Plain old sequential

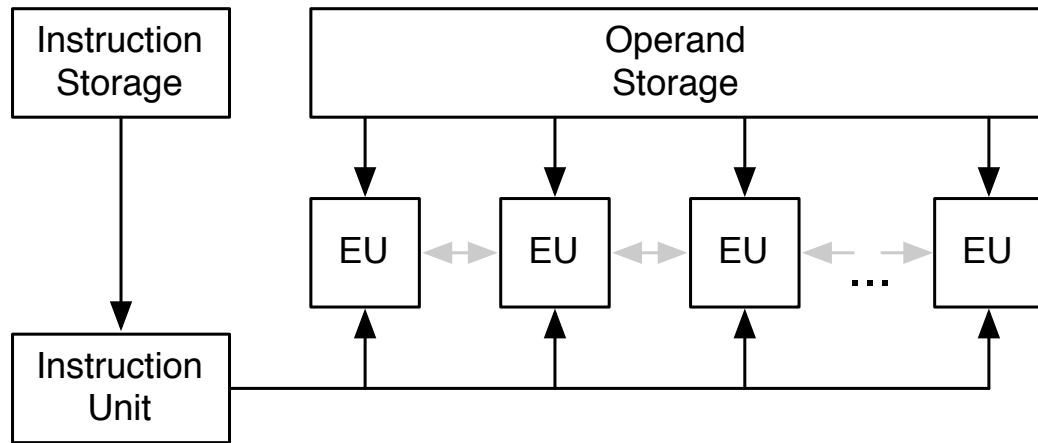
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD



Based on multiplicity of instruction streams, data storage



SIMD in SSE/AVX



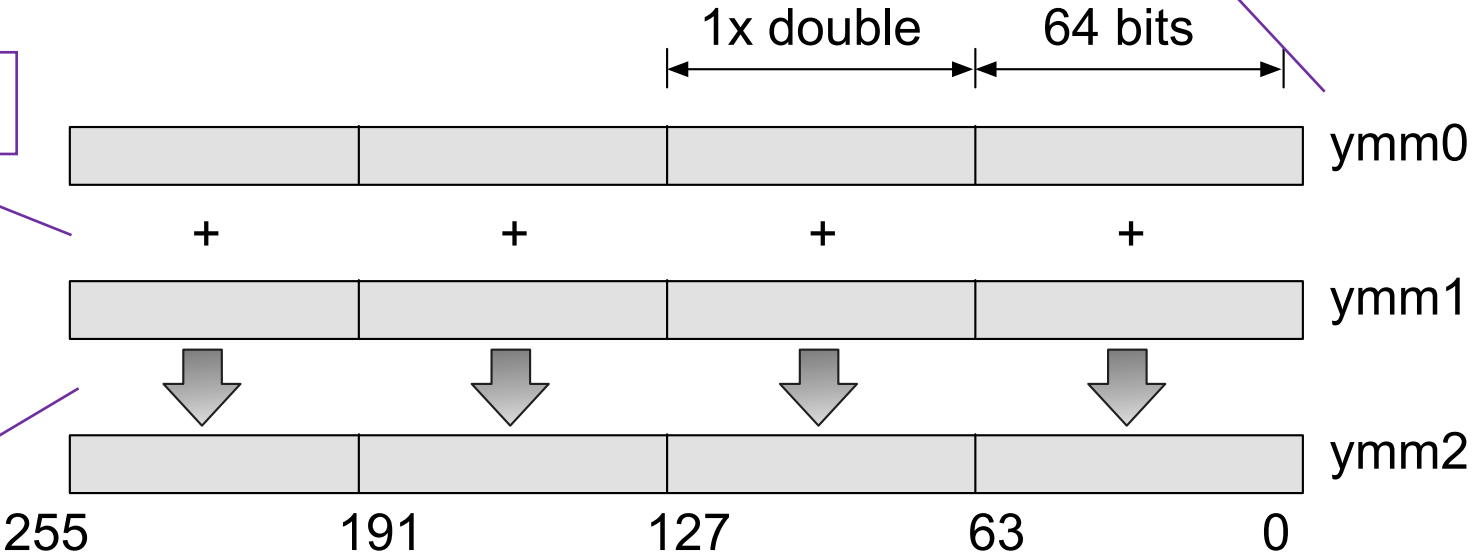
Flynn's original conceptual model

ymm are 256 bit registers

```
vfadd231pd %ymm0, %ymm1, %ymm2
```

One machine instruction

Adds all four doubles *simultaneously*



SIMD and MIMD

- Two principal parallel computing paradigms (multiple op

Single instruction at a time

Multiple instructions

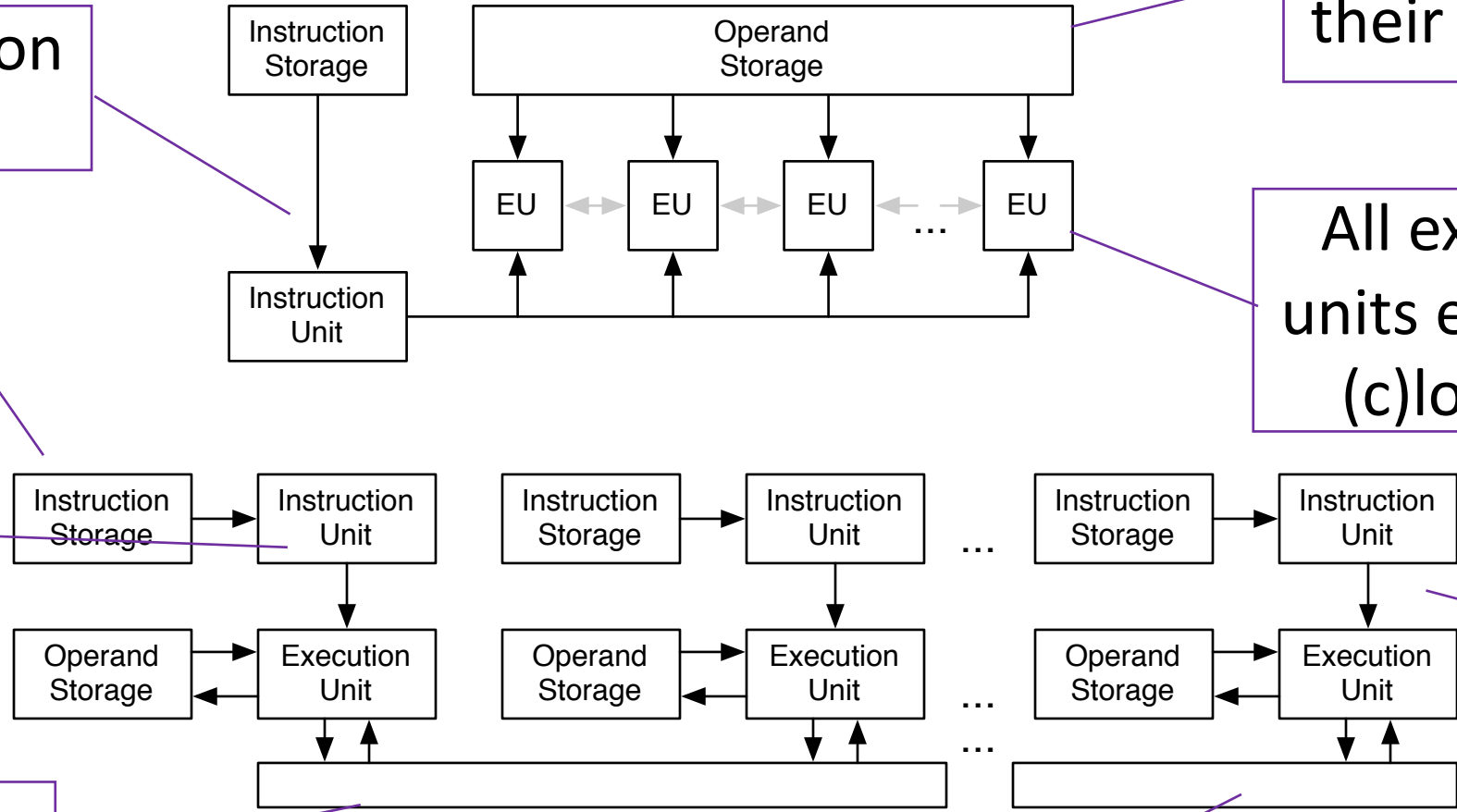
EUs run independently (w own instrs)

Shared Memory

But each have their own data

All execution units execute in (c)lock step

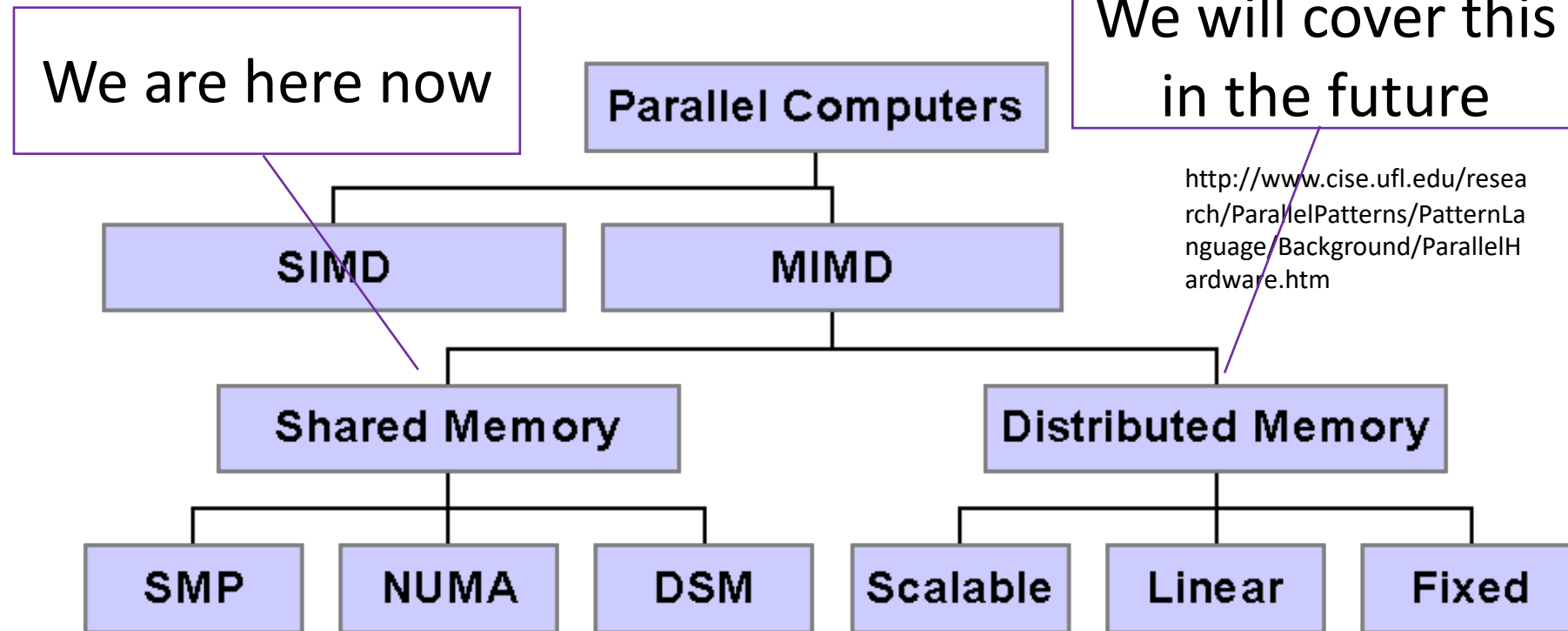
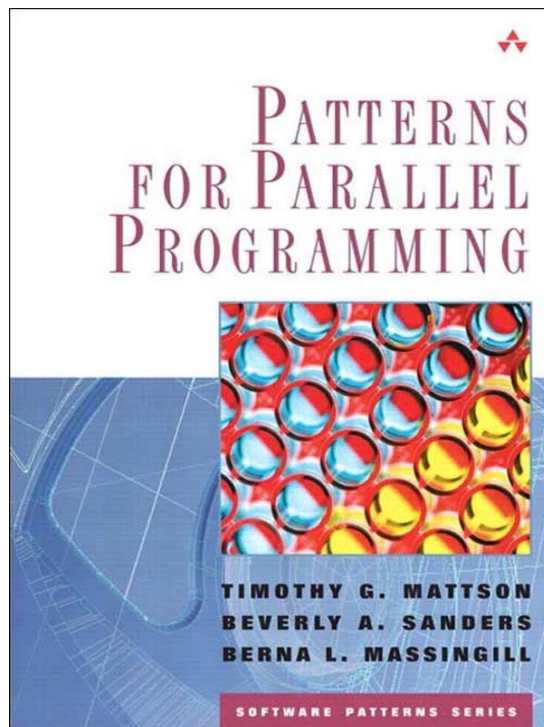
Coming up next



Not Shared

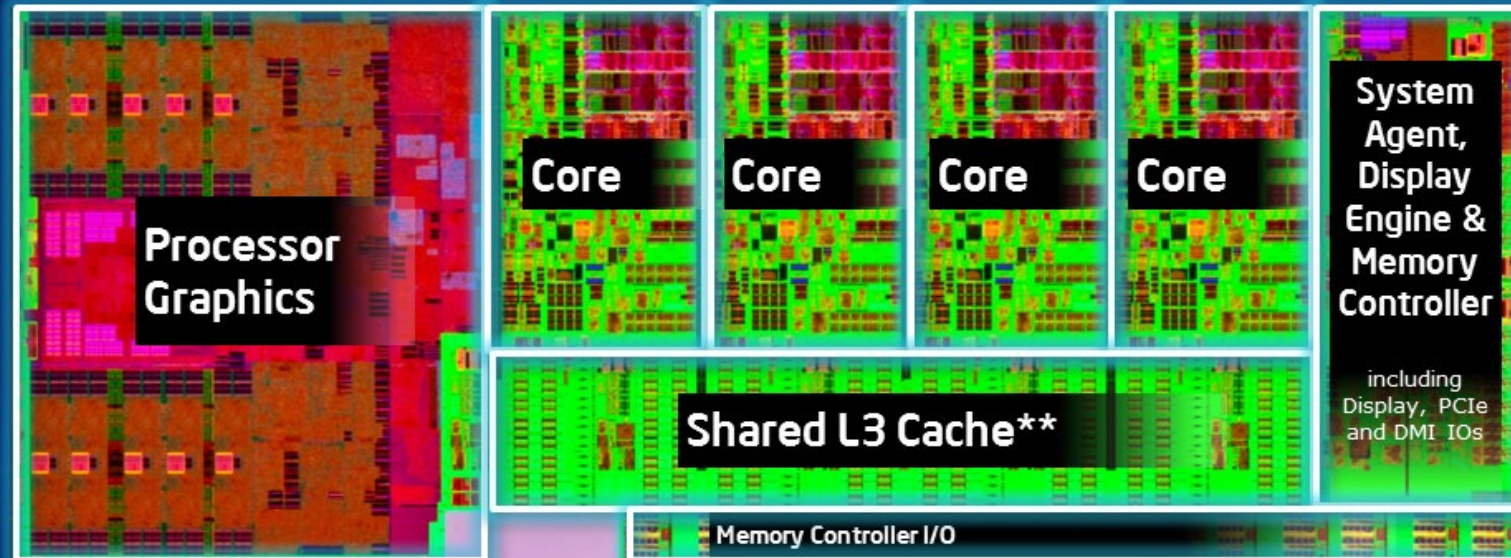
A More Refined (Programmer-Oriented) Taxonomy

- Three major modes: SIMD, Shared Memory, Distributed Memory
- Different programming approaches are generally associated with different modes of parallelism (threads for shared, MPI for distributed)
- A modern supercomputer will have all three major modes present



Multicore Architecture

4th Generation Intel® Core™ Processor Die Map 22nm Haswell Tri-Gate 3-D Transistors



Quad core die shown above | Transistor count: 1.4Billion | Die size: 177mm²

** Cache is shared across all 4 cores and processor graphics

All products, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

UNDER EMBARGO UNTIL FURTHER NOTICE

INTEL CONFIDENTIAL



Multicore for HPC

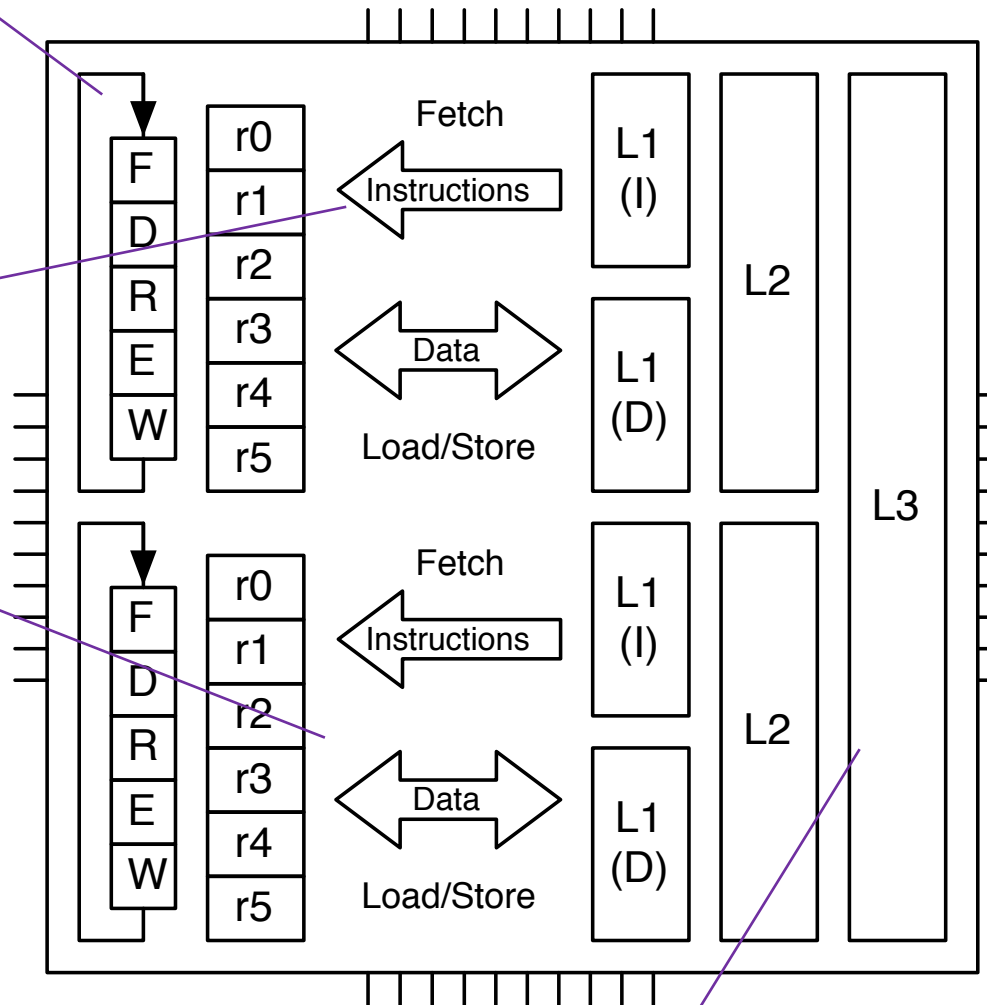
- How do multicore chips operate (how does the hardware work)?
- How do they get high performance?
- How does the software exploit the hardware (how do we write our software to exploit the hardware)?
- What are the abstractions that we need to use to reason about multicore systems?
- What are the programming abstractions and mechanisms?
- Terminology: Program, process, thread
- More terminology: Parallel, concurrent, asynchronous

Multicore Architecture

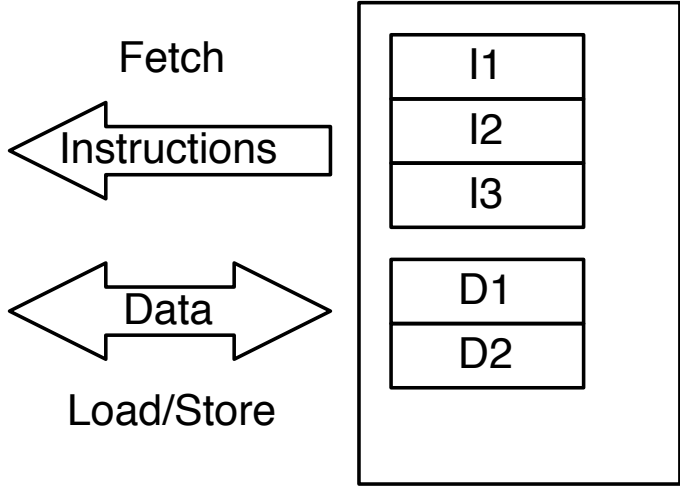
Core is a
FDREW + regs

Each runs its
own sequence
of instructions

Each can access
its own data



Any CPU in the
last 4-5 years



Each has memory
hierarchy

But memory
might be shared

Sequential Example

- You are the TA for AMATH 483 and must grade 22 exams
- The exam has 8 questions on it
- It takes 3 minutes to grade one question
- How long will it take you to grade all the exams?



Parallelization Example

- You are the TA for AMATH 483 and must grade 22 exams
- The exam has 8 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you
- How long will it take the 22 of you to grade all the exams?
- Describe your approach
- List your assumptions



Parallelization Example

- You are the TA for AMATH 483 and must grade 1012 exams ($1012 = 46 * 22$)
- The exam has 8 questions on it
- It takes 3 minutes to grade one question
- You ask 21 friends who agree to help you
- How long will it take the 22 of you to grade all the exams?
- Describe your approach
- Describe another approach
- List your assumptions



Another Parallelization Example

- You are the TA for AMATH 483 and must grade 8 exams
 - The exam has 22 questions on it
 - It takes 3 minutes to grade one question
 - You ask 21 friends who agree to help you
-
- How long will it take the 22 of you to grade all the exams?
-
- Describe your approach



Parallelization Example

- You are the TA for AMATH 483 and must grade 368 exams ($368 = 46 * 8$)
 - The exam has 22 questions on it
 - It takes 3 minutes to grade one question
 - You ask 21 friends who agree to help you
-
- How long will it take the 22 of you to grade all the exams?
-
- What if you had 368 friends? $368 * 22$?

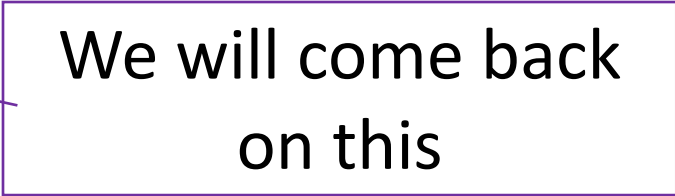


Compare And Contrast – Concurrency vs Parallelism

- Time for everyone grades one exam: $8 * 3$ minutes
- Time for everyone grades one question: 3 minutes

- How (why) did you use the approaches you did?

- Concurrency
 - When two or more tasks can start, run, and complete in overlapping time periods
- Parallelism
 - When tasks **literally** run at the same time



We will come back
on this

How Do We Run Many Programs at the Same Time?

The screenshot displays a macOS desktop environment with several overlapping windows:

- Code Editor (Matrix.cpp):** Shows C++ code for matrix multiplication. The visible code includes:

```
void hoistedMultiply(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); ++i) {  
        for (int j = 0; j < B.numCols(); ++j) {  
            double t = C(i,j);  
            for (int k = 0; k < A.numCols(); ++k) {  
                t += A(i,k) * B(k,j);  
            }  
            C(i,j) = t;  
        }  
    }  
}  
  
void tiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
            }  
        }  
    }  
}  
  
void tiledMultiply2x4(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 4) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i, j+2) += A(i, k) * B(k, j+2);  
                C(i, j+3) += A(i, k) * B(k, j+3);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
                C(i+1, j+2) += A(i+1, k) * B(k, j+2);  
                C(i+1, j+3) += A(i+1, k) * B(k, j+3);  
            }  
        }  
    }  
}  
  
void tiledMultiply4x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 4) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
            }  
        }  
    }  
}
```
- Terminal Window:** Shows a series of shell commands being executed in a directory named L8:

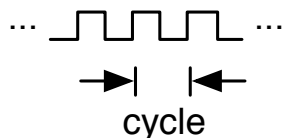
```
lums658@WE31821=> cd ..  
lums658@WE31821=> cp L7/L7.pptx L8  
lums658@WE31821=> cd L8  
lums658@WE31821=> mv L7.pptx L8.pptx  
lums658@WE31821=> open L8.pptx  
lums658@WE31821=> ls  
L8.pptx  
lums658@WE31821=> git add L8.pptx  
lums658@WE31821=>
```
- Web Browser:** A search for "douglas adams" is shown on the Google search page.

Running a Program

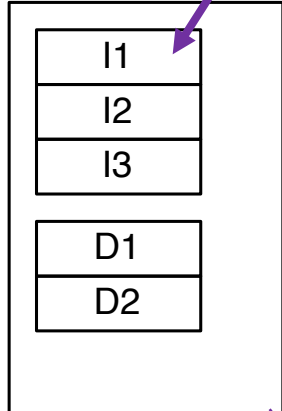
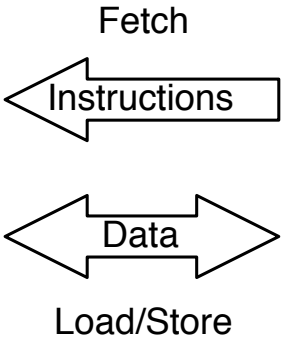
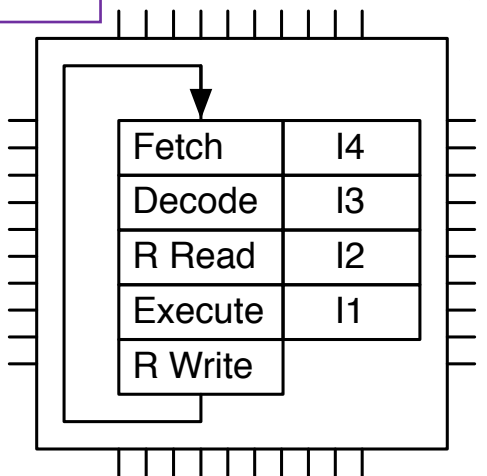
When a CPU is executing bytes from one program

Bytes from program stored in memory

It isn't executing bytes from another



Including from the OS (just another program)



```

.globl    __Z15hoistedMultiplyRK6MatrixS1_RS_
.p2align 4, 0x90
__Z15hoistedMultiplyRK6MatrixS1_RS_:    ## @_Z15hoistedMultiplyRK6MatrixS1_RS_
.cfi_startproc
## BB#0:
    pushq   %rbp
Ltmp16:
.cfi_def_cfa_offset 16
Ltmp17:
.cfi_offset %rbp, -16
    movq   %rsp, %rbp
Ltmp18:
.cfi_def_cfa_register %rbp
    pushq  %r15
    pushq  %r14
    pushq  %r13
    pushq  %r12
    pushq  %rbx
Ltmp19:
.cfi_offset %rbx, -56
Ltmp20:
.cfi_offset %r12, -48
Ltmp21:
.cfi_offset %r13, -40
Ltmp22:
.cfi_offset %r14, -32
Ltmp23:
.cfi_offset %r15, -24
    movq   (%rdi), %rax
    movq   %rax, -120(%rbp)    ## 8-byte Spill
    testq  %rax, %rax
    je     LBB2_9
## BB#1:
    movq   8(%rsi), %rcx
    testq  %rcx, %rcx
    je     LBB2_9
## BB#2:
    movq   16(%rsi), %r12
    movq   8(%rdx), %rax
    movq   %rax, -104(%rbp)    ## 8-byte Spill
    movq   16(%rdx), %rdx
    movq   8(%rdi), %rax
    movq   16(%rdi), %r13
    leaq  -1(%rcx), %rsi
    movq   %rsi, -88(%rbp)    ## 8-byte Spill
    movl   %ecx, %esi
    
```

How does another program run?

How did the bytes get here?

How Do We Run Many Programs “at the Same Time”?

The screenshot displays a Mac desktop environment with several overlapping windows:

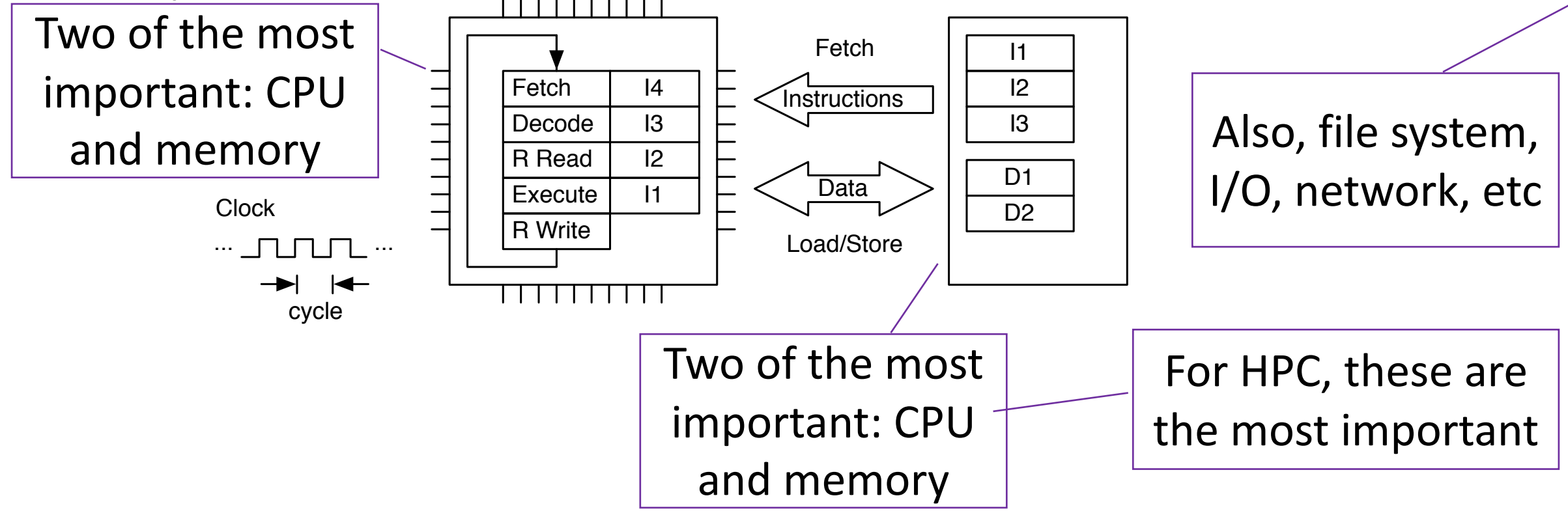
- Code Editor (Matrix.cpp):** Shows C++ code for matrix multiplication. The visible code includes:

```
void hoistedMultiply(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); ++i) {  
        for (int j = 0; j < B.numCols(); ++j) {  
            double t = C(i,j);  
            for (int k = 0; k < A.numCols(); ++k) {  
                t += A(i,k) * B(k,j);  
            }  
            C(i,j) = t;  
        }  
    }  
}  
  
void tiledMultiply2x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
            }  
        }  
    }  
}  
  
void tiledMultiply2x4(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 2) {  
        for (int j = 0; j < B.numCols(); j += 4) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i, j+2) += A(i, k) * B(k, j+2);  
                C(i, j+3) += A(i, k) * B(k, j+3);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
                C(i+1, j+2) += A(i+1, k) * B(k, j+2);  
                C(i+1, j+3) += A(i+1, k) * B(k, j+3);  
            }  
        }  
    }  
}  
  
void tiledMultiply4x2(const Matrix& A, const Matrix&B, Matrix&C) {  
    for (int i = 0; i < A.numRows(); i += 4) {  
        for (int j = 0; j < B.numCols(); j += 2) {  
            for (int k = 0; k < A.numCols(); ++k) {  
                C(i, j) += A(i, k) * B(k, j);  
                C(i, j+1) += A(i, k) * B(k, j+1);  
                C(i+1, j) += A(i+1, k) * B(k, j);  
                C(i+1, j+1) += A(i+1, k) * B(k, j+1);  
            }  
        }  
    }  
}
```
- Terminal Window:** Shows shell commands for navigating to a directory and running a program:

```
lums658@WE31821 - /Users/lums658/git/amath-583/lectures/L8 — lums658@WE31821 — tcsh — 148x64  
lums658@WE31821=> cd ..  
lums658@WE31821=> cp L7/L7.pptx L8  
lums658@WE31821=> cd L8  
lums658@WE31821=> mv L7.pptx L8.pptx  
lums658@WE31821=> open L8.pptx  
lums658@WE31821=> ls  
L8.pptx  
lums658@WE31821=> git add L8.pptx  
lums658@WE31821=> █
```
- Web Browser:** Shows a Google search for "douglas adams".
- Background Windows:** A presentation window titled "piazza" and a window titled "Seattle Maker Space" are visible.

A Word About Operating Systems

- An operating system is **a (system) program** that provides a standard interface between the resources of a computer and the users of the computer



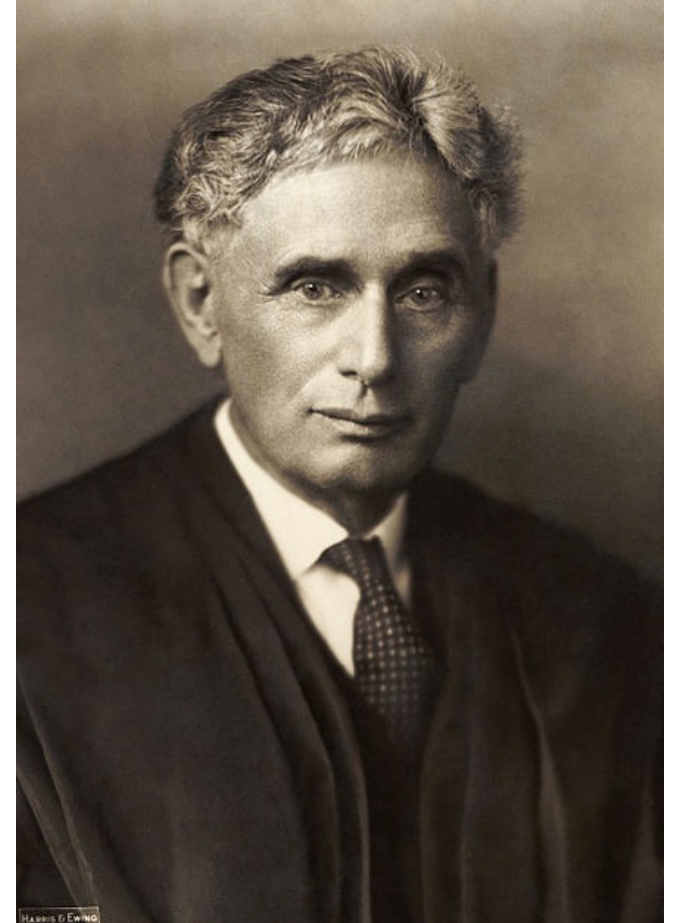
How Do We Run Many Programs Concurrently?

The screenshot shows a multi-tasking environment. In the foreground, a code editor displays C++ code for matrix multiplication. The code includes a standard `multiply` function and three tiled versions: `hoistedMultiply`, `tiledMultiply2x2`, and `tiledMultiply4x2`. The `hoistedMultiply` function uses a `double t` variable to store intermediate results. The tiled versions use nested loops to process blocks of the matrix. A terminal window in the background shows the execution of these programs, with commands like `cd ..`, `cp L7/L7.pptx L8`, `mv L7.pptx L8.pptx`, `open L8.pptx`, `ls`, `git add L8.pptx`, and `git commit -m 'add L8.pptx'`. A web browser in the background shows a search for 'douglas adams' on Google.

Do not *ever* say: "the operating system stops the first process and starts the next"

Processes

- A process is an abstraction for a collection of resources to represent a (running) program
 - CPU
 - Memory
 - Address space



The Operating System Can Run When...

- The process whose instructions are being executed by the CPU (the running process) requests a service from the OS (makes a ***system call***)
- In response to a hardware interrupt
- It does not spontaneously run
- It is not somehow running in the background
- Again, when the CPU is executing instructions for one program, it is not executing instructions for another program
- The only way anything happens on the computer is if the CPU executes instructions that make it happen

Process Abstraction

Stored in Process Control Block (PCB)

Set of information about process resources

Sufficient to be able to start a process after stopped

Also for accounting / administrative purposes

Process management

Registers
Program counter
Program status word
Stack pointer
Process state
Priority
Scheduling parameters
Process ID
Parent process
Process group
Signals
Time when process started
CPU time used
Children's CPU time
Time of next alarm

Memory management

Pointer to text segment
Pointer to data segment
Pointer to stack segment

File management

Root directory
Working directory
File descriptors
User ID
Group ID

What does program counter represent?

The Process Concept

\$ top -u

Process ID

How much CPU

How many threads

```
lums658@WE31821 - /Users/lums658
Processes: 419 total, 2 running, 417 sleeping, 1988 threads
Load Avg: 1.93, 1.88, 1.87 CPU usage: 3.45% user, 3.69% sys,
MemRegions: 156549 total, 7076M resident, 141M private, 3629M
VM: 4328G vsize, 627M framework vsize, 71344832(64) swapins,
Disks: 57070556/1524G read, 36025949/792G written.
```

```
Process: WindowServer (13.8)
Load Avg: 1.93, 1.88, 1.87 CPU usage: 3.45% user, 3.69% sys, 92.84% idle SharedLibs: 252M resident, 48M data, 60M linkedin.
MemRegions: 156549 total, 7076M resident, 141M private, 3629M shared. PhysMem: 16G used (2616M wired), 236M unused.
VM: 4328G vsize, 627M framework vsize, 71344832(64) swapins, 74484796(0) swapouts. Networks: packets: 41299644/296 in, 41044343/260 out.
Disks: 57070556/1524G read, 36025949/792G written.

PID COMMAND %CPU TIME #TH #WQ #PORT MEM PURG CMPRS PGRP PPID STATE BOOSTS
0 kernel_task 12.6 29:59:12 177/9 0 2 1809M+ 0B 0B 0 0 running *0[1]
114 hidd 4.4 01:46:55 6 3 281 3024K+ 0B 1368K 114 1 sleeping *0[1]
8333 top 4.0 00:00.72 1/1 0 21 5016K 0B 0B 8333 67567 running *0[1]
8334 screencaptur 3.9 00:00.06 4 3 57 2500K+ 20K 0B 853 853 sleeping *0[1]
91791 LaTeXiT 2.3 09:45.97 6 2 255 42M 0B 30M 91791 1 sleeping *0[113]
67565 Terminal 2.0 01:50:53 13 8 346+ 72M 0B 19M 67565 1 sleeping *0[1555]
3288 Calendar 1.6 09:54.07 3 1 292 95M 1856K 39M 3288 1 sleeping *0[3352]
1234 com.docker.h 1.1 02:02:24 18 1 38 763M 0B 487M 1228 1228 sleeping *0[1]
846 usernoted 1.1 03:13:97 5 4 139+ 11M+ 896K 0B 868K 846 1 sleeping *0[1]
83898 Slack Helper 1.0 01:40:81 19 2 149 189M+ 0B 49M 63333 63333 sleeping *0[4]
91742 splunkd 0.8 40:02:25 35 0 48 85M 0B 47M 91741 1 sleeping *0[1]
63334 Slack Helper 0.6 01:19:70 5 2 124 7780K 0B 26M 63333 63333 sleeping *0[1]
184 mDNSResponse 0.5 22:51.68 5 1 103 5628K 0B 3408K 184 1 sleeping *0[1]
111- NetworkMonit 0.4 12:37.75 28 27 49+ 22M+ 0B 185M 111 1 sleeping *0[1]
883 CalNCService 0.3 19:18.74 5 3 182+ 39M+ 0B 5184K 883 1 sleeping *2680[2102]
853 SystemUIServ 0.2 02:45.35 5 3 371 33M+ 28K- 25M 853 1 sleeping *0[2993]
63333 Slack 0.2 04:36.66 33 1 390 73M 0B 26M 63333 63333 sleeping *0[803]
214 com.apple.if 0.1 16:08.11 5 3 381 1760K 0B 1256K 214 1 sleeping *0[10988]
42449 Notification 0.1 01:22.32 5 2 295+ 34M+ 756K 32M 42449 1 sleeping *0[13680+]
1163- netsession_m 0.1 16:44.56 9 1 43 2924K 0B 6796K 1163 1 sleeping *0[1]
1173 java 0.0 13:22.66 27 1 218 26M 0B 116M 1173 1 sleeping *0[1]
93 SymDaemon 0.0 63:53.97 20 4 154 316M 0B 61M 93 1 sleeping *0[1]
89822 Microsoft Po 0.0 12:19.35 22 5 489 1172M 117M 183M 89822 1 sleeping *0[1162]
53- dsAccessServ 0.0 07:36.73 14 5 115 2376K 0B 2912K 53 1 sleeping *0[1]
818 ComCenter 0.0 01:14.31 8 3 264 3960K 0B 4860K 818 1 sleeping *0[1]
1225 com.docker.o 0.0 02:05.83 11 1 49 196K 0B 16M 1225 1221 sleeping *0[1]
1157 CrashPlanWeb 0.0 78:13.14 27 2 330 47M 12K 36M 1157 1 sleeping *0[1]
8331 SChelper 0.0 00:00.01 3 2 28+ 644K+ 0B 0B 8331 1 sleeping *0[13]
1147 AOUonitor 0.0 04:32.71 8 2 175 15M 0B 10M 1147 1 sleeping *0[1]
60 logd 0.0 13:28.40 4 4 823 36M 0B 11M 60 1 sleeping *0[1]
422 CrashPlanSer 0.0 42:24:56 89 1 236 22M 0B 458M 42 1 sleeping *0[0]
124 com.docker.d 0.0 04:26.05 8 0 17 21M 0B 12M 1224 1221 sleeping *0[1]
814 UserEventAge 0.0 01:50.69 3 1 632 4512K 0B 2112K 814 1 sleeping *0[1]
42572 Jabra Skype 0.0 03:04.56 4 2 154 3936K 0B 83M 42572 1 sleeping *0[1375]
84167 Slack Helper 0.0 00:31.70 20 2 150 148M 0B 32M 63333 63333 sleeping *0[4]
66996 com.apple.We 0.0 02:52.26 5 1 144 34M 0B 1328K 66996 1 sleeping *2[1]
64 airportd 0.0 10:06.49 3 1 383 24M 0B 11M 64 1 sleeping *322[711]
69950 Mail 0.0 17:22.00 10 3 549 242M 29M 56M 69950 1 sleeping *0[3148]
1024 sharingd 0.0 03:40.28 4 1 235 23M 0B 7900K 1024 1 sleeping *461588]
1121 SafariCloudH 0.0 03:28.67 4 3 48 1598K 0B 948K 1121 1 sleeping *0[1]
860 cloudphotosd 0.0 01:10.67 6 1 273 5320K 0B 20M 860 1 sleeping *0[5239]
102 blues 0.0 00:01.89 3 1 167 7608K 0B 1608K 102 1 sleeping *0[1]
1155 SymUIAgent 0.0 01:09.37 5 1 195 6500K 0B 11M 1155 1 sleeping *0[1]
186 mDNSResponse 0.0 00:33.85 3 2 60 1688K 0B 932K 186 1 sleeping *0[1]
116 AirPlayXPCh 0.0 00:15.07 2 2 131 2024K 0B 2676K 116 1 sleeping *0[1]
218 symptomsd 0.0 02:34.93 3 2 165 5104K 0B 4088K 218 1 sleeping *0[47812]
89454 ntpd 0.0 00:01.40 3 3 28 776K 0B 1528K 89454 1 sleeping *0[1]
97 locationd 0.0 02:25.93 6 1 125 8276K 256K 4644K 97 1 sleeping *0[81714]
52 configd 0.0 00:51.25 11 4 635 609K+ 0B 6520K 52 1 sleeping *0[1]
6167 mdworker 0.0 00:00.77 4 1 54 2944K 0B 22M 6167 1 sleeping *0[1]
47- vpnagentd 0.0 06:20.57 6 1 64 6488K 0B 18M 47 1 sleeping *0[1]
195 mds_stores 0.0 03:00:12 6 4 117 183M 2188K 46M 195 1 sleeping *0[1]
861 CalendarAgen 0.0 02:43:54 3 1 334 80M 28M 48M 861 1 sleeping *0[614163]
205 coreaudioid 0.0 01:52:15 3 1 347 3112K 0B 2268K 205 1 sleeping *0[1]
1382 Electron Hel 0.0 91:49.35 19 2 113 68M 0B 28M 1157 1 sleeping *0[1]
67 mds 0.0 00:27.80 9 4 920 61M 0B 52M 67 1 sleeping *0[1]
```

PID	COMMAND	%CPU	TIME	#TH	#WQ	#PORT	MEM	PURG
162	WindowServer	13.8	07:48:22	6	2	702+	537M+	93M
0	kernel_task	12.6	29:59:12	177/9	0	2	1809M+	0B
4	hidd	4.4	01:46:55	6	3	381+	3024K+	0B
33	top	4.0	00:00.72	1/1	0	21	5016K	0B
34	screencaptur	3.9	00:00.06	4	3	57	2500K+	20K
791	LaTeXiT	2.3	09:45.97	6	2	255	42M	0B
67565	Terminal	2.0	01:50:53	13	8	346+	72M	0B
3288	Calendar	1.6	09:54.07	3	1	292	95M	185K
1234	com.docker.h	1.1	02:02:24	18	1	38	763M	0B
846	usernoted	1.1	03:13:97	5	4	139+	11M+	896K
83898	Slack Helper	1.0	01:40:81	19	2	149	189M+	0B
91742	splunkd	0.8	40:02.25	35	0	48	85M	0B
63334	Slack Helper	0.6	01:19:70	5	2	124	7780K	0B
184	mDNSResponse	0.5	22:51.68	5	1	103	5628K	0B
111-	NetworkMonit	0.4	12:37.75	28	27	49+	22M+	0B
883	CalNCService	0.3	19:18.74	5	3	182+	39M+	0B
853	SystemUIServ	0.2	02:45.35	5	3	371	33M+	28K-
63333	Slack	0.2	04:36.66	33	1	390	73M	0B

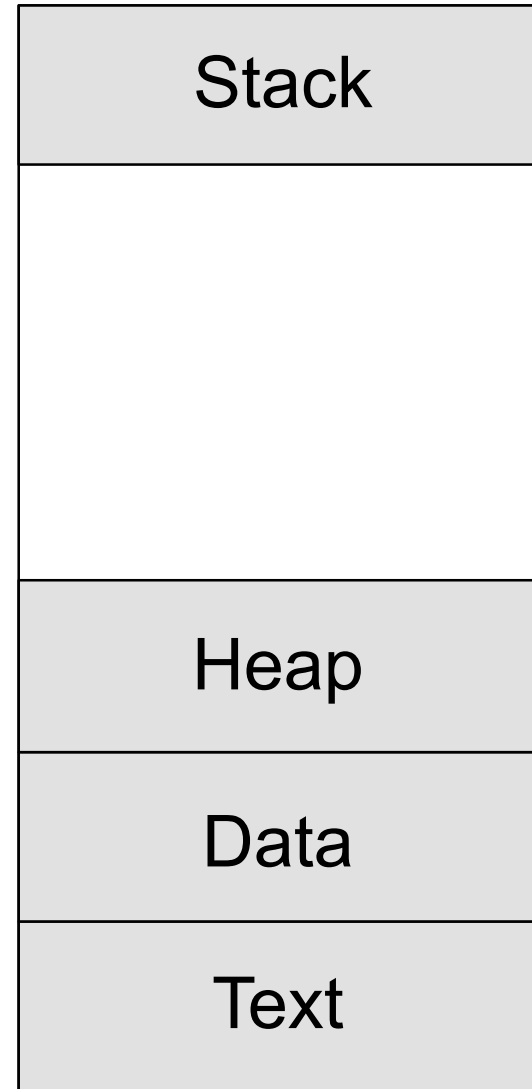
Process address space

Memory resources
for each process

All 32/48/64 bits

Address
Space

How can each
process use all the
address space?



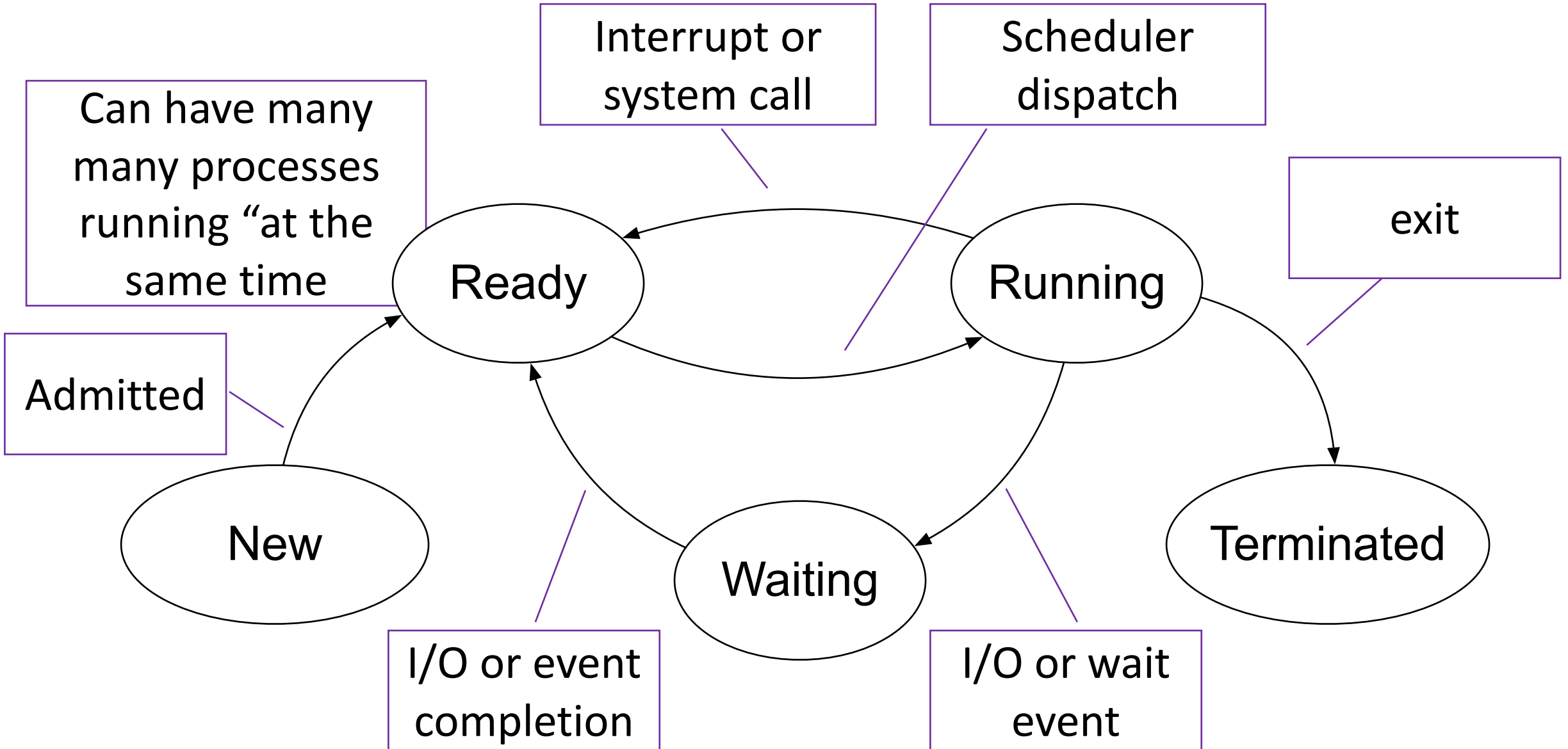
Created and
managed at run time

Created and
managed at run time

Compiled /
Linked

Stored
Program

Process Lifetime



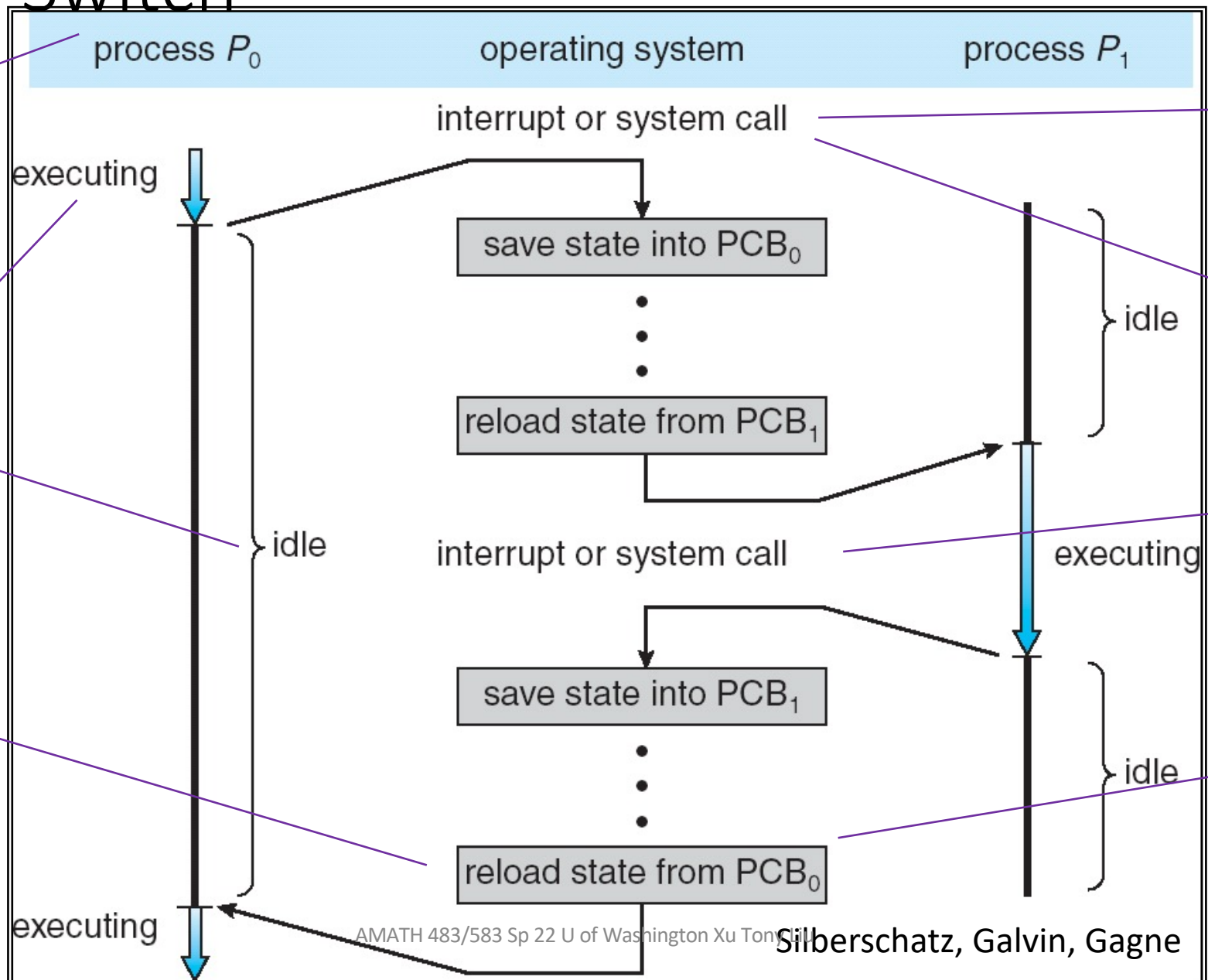
Context Switch

P0 and P1 are running processes

What does this mean?

And this?

PCB = Process Control Block



External to OS

OS does not do this

External to OS

Expensive!

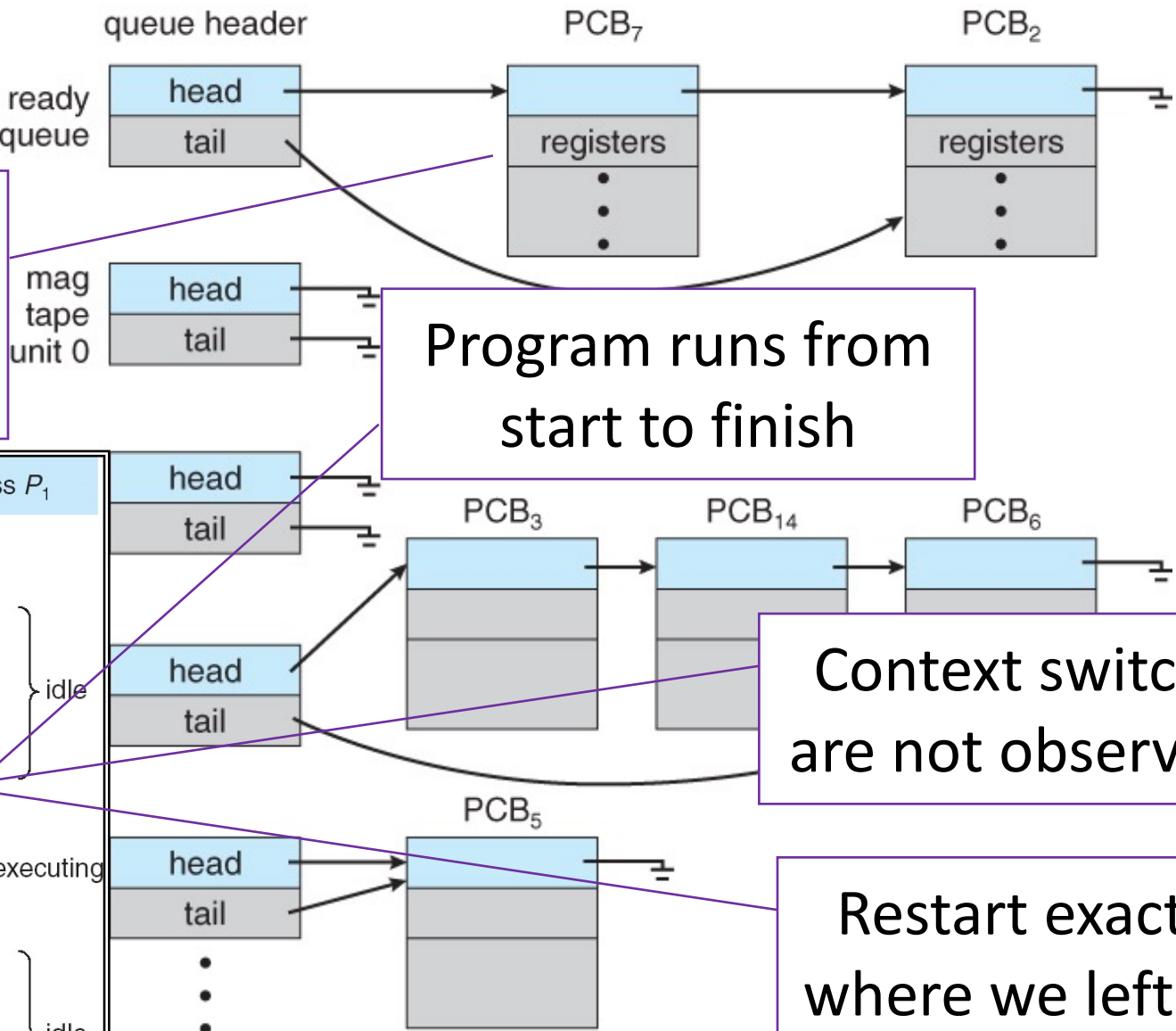
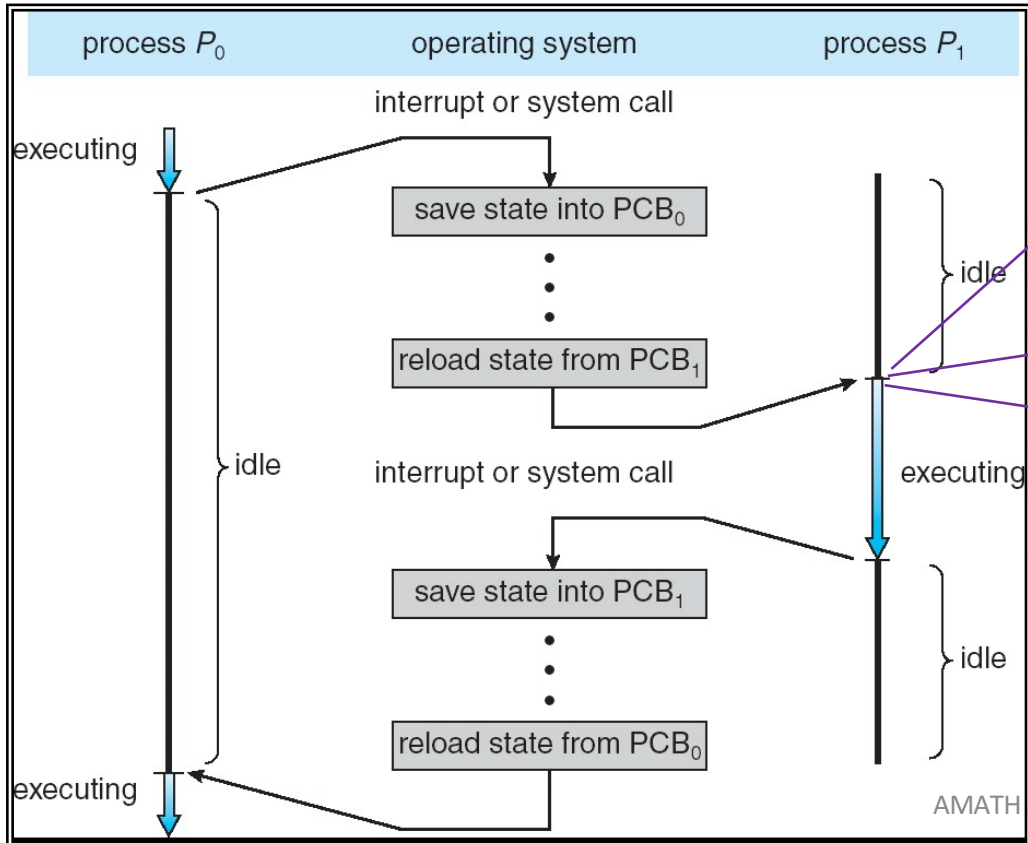
Process Queues

A process control block (PCB) has all information necessary to manage a process

Program runs from start to finish

Context switches are not observable

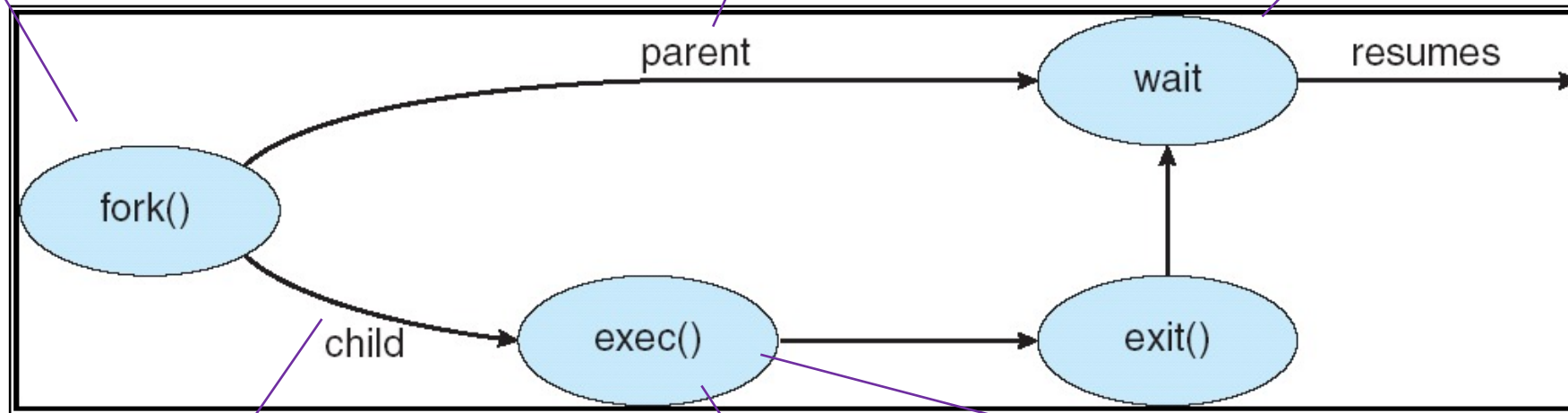
Restart exactly where we left off



Process invokes fork()

The other process (the "parent") keeps executing

Can wait for other process to complete



The OS makes a copy of the original process and makes it runnable

One of the processes (the "child") runs exec()

Which pulls in new program bits to run

You see this fork/exec/wait almost all the time with one particular program you run (which?)

Example: process creation in UNIX

One process calls fork()

```
#include <unistd.h>

int main () {
    fork();
    return 0;
}
```

Each process "thinks" it called fork() and returned

Two processes return from fork()

Two processes return from fork()

```
#include <unistd.h>

int main () {
    fork();
    return 0;
}
```

```
#include <unistd.h>

int main () {
    fork();
    return 0;
}
```

fork() make an exact copy

Example

fork() returns a
PID identifier

Loop 20 times

Call fork() 20
times

How many processes
get created?

```
int main() {  
  {  
    int pids[20];  
  
    for (int i = 0; i < 20; ++i) {  
      pids[i] = fork();  
    }  
  
    return 0;  
  }  
}
```

Example

How deep is the tree?

How many processes?

Don't do this (ever)!

$i == 0$

$i == 1$

$i == 2$

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
int main() {
{
  int pids[20];

  for (int i = 0; i < 20; ++i) {
    pids[i] = fork();
  }

  return 0;
}
```

```
man fork()
#include <unistd.h>
pid_t fork();
```

The child process has a unique id

Upon successful completion, fork() returns a value of 0 to the child process and the returns the process ID of the child process to the parent process

```
BSD System Calls Manual
NAME
fork -- create a new process
SYNOPSIS
#include <unistd.h>
pid_t
fork(void);
DESCRIPTION
fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:
o The child process has a unique process ID.
o The child process has a different parent process ID (i.e., the process ID of the parent process).
o The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
o The child processes resource utilizations are set to 0; see setrlimit(2).
RETURN VALUES
Upon successful completion, fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.
ERRORS
fork() will fail and no child process will be created if:
[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. This limit is configuration-dependent.
[EAGAIN] The system-imposed limit MAXUPRC (<sys/param.h>) on the total number of processes under execution by a single user would be exceeded.
[ENOMEM] There is insufficient swap space for the new process.
EXAMPLES
SYNOPSIS
#include <sys/types.h>
#include <unistd.h>
The include file <sys/types.h> is necessary.
SEE ALSO
```

Example Revisited

```
int main() {  
    {  
        pid_t pids[20];  
  
        for (int i = 0; i < 20; ++i) {  
            pids[i] = fork();  
            if (pids[i] == 0)  
                break;  
        }  
  
        return 0;  
    }  
}
```

Get return
value of fork()

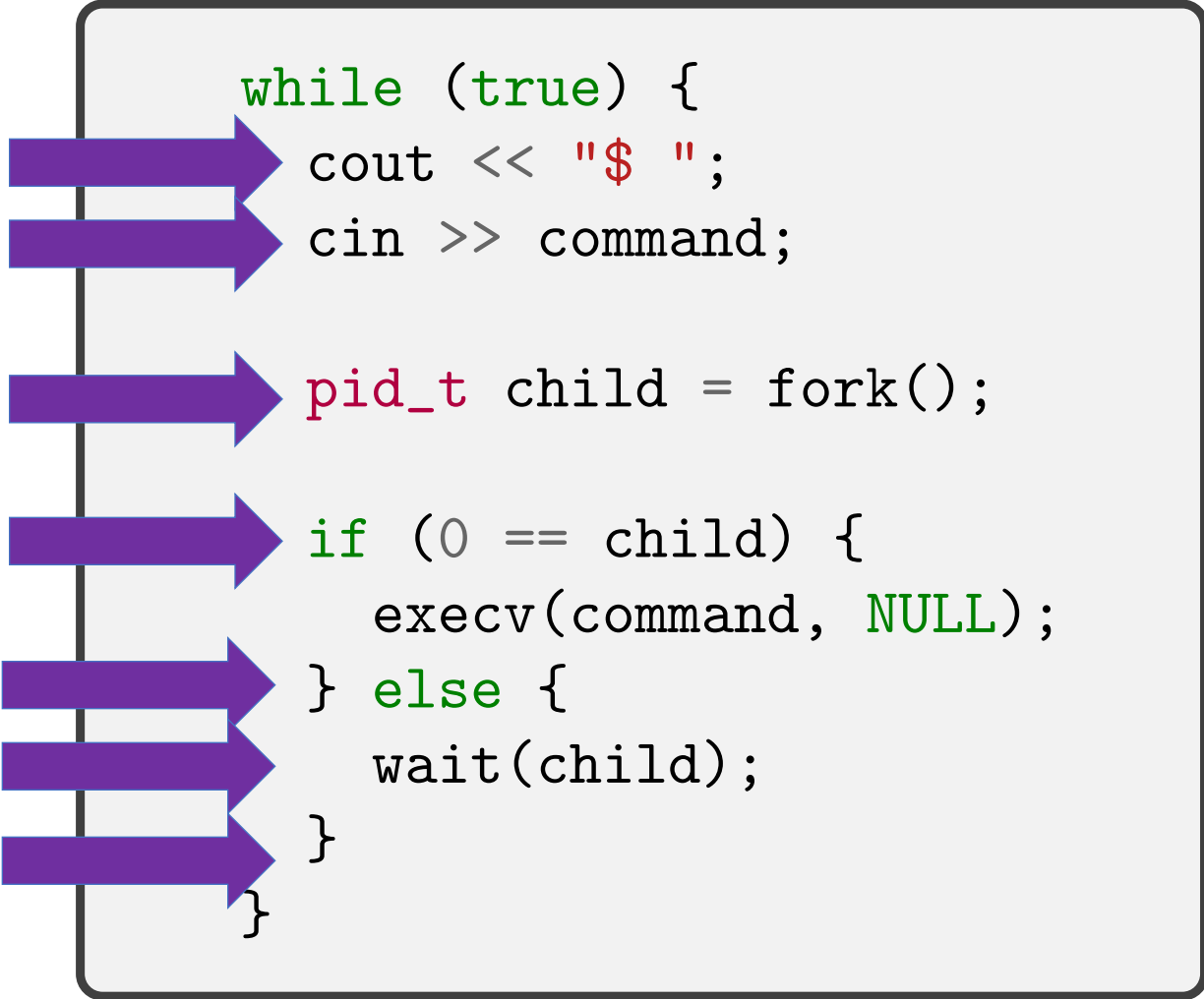
How many
processes
now?

If zero, the
process is a child

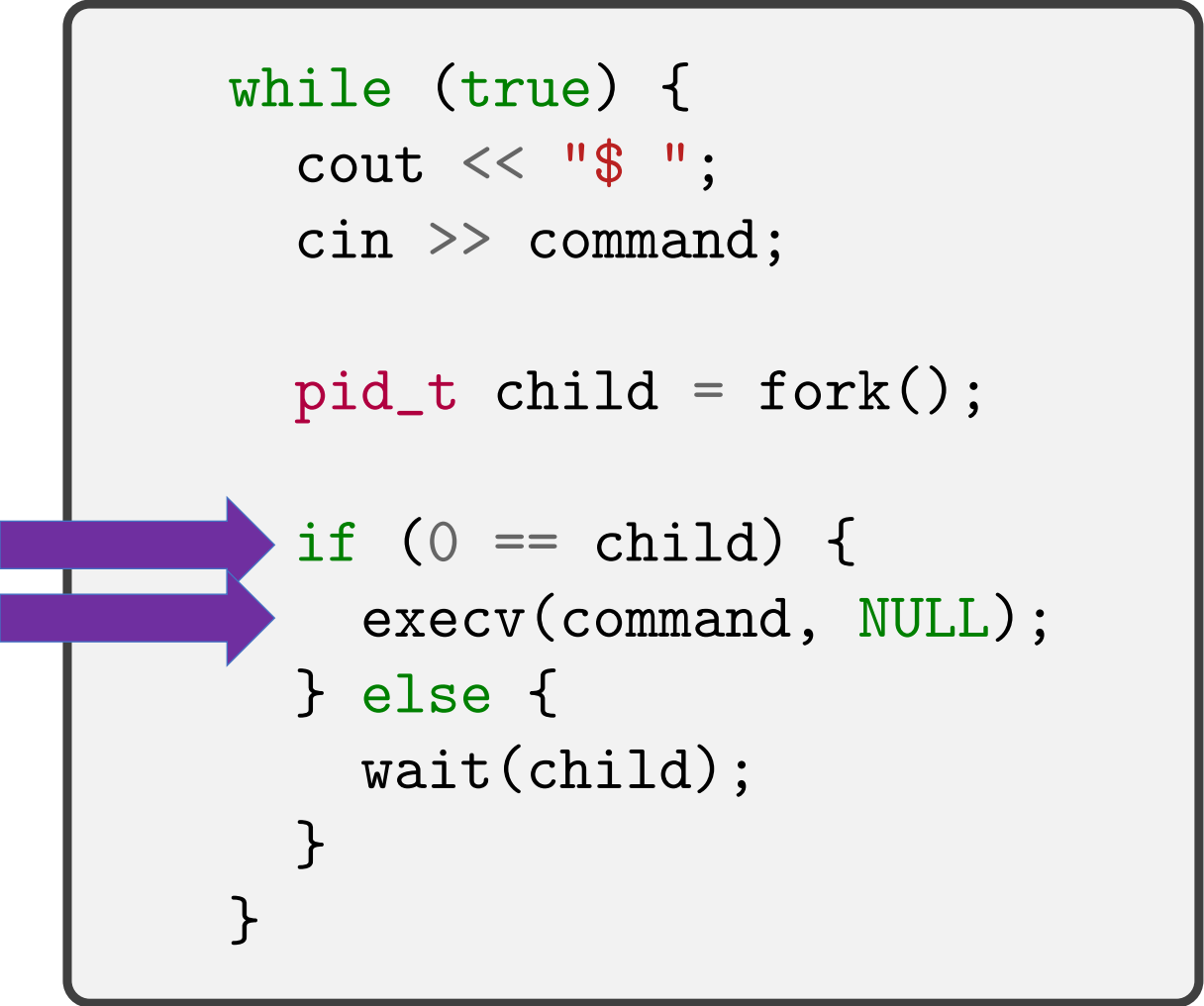
If no, the process
is the parent,
keep going

Process creation in UNIX (fork / exec pattern)

```
while (true) {  
    cout << "$ ";  
    cin >> command;  
    pid_t child = fork();  
    if (0 == child) {  
        execv(command, NULL);  
    } else {  
        wait(child);  
    }  
}
```

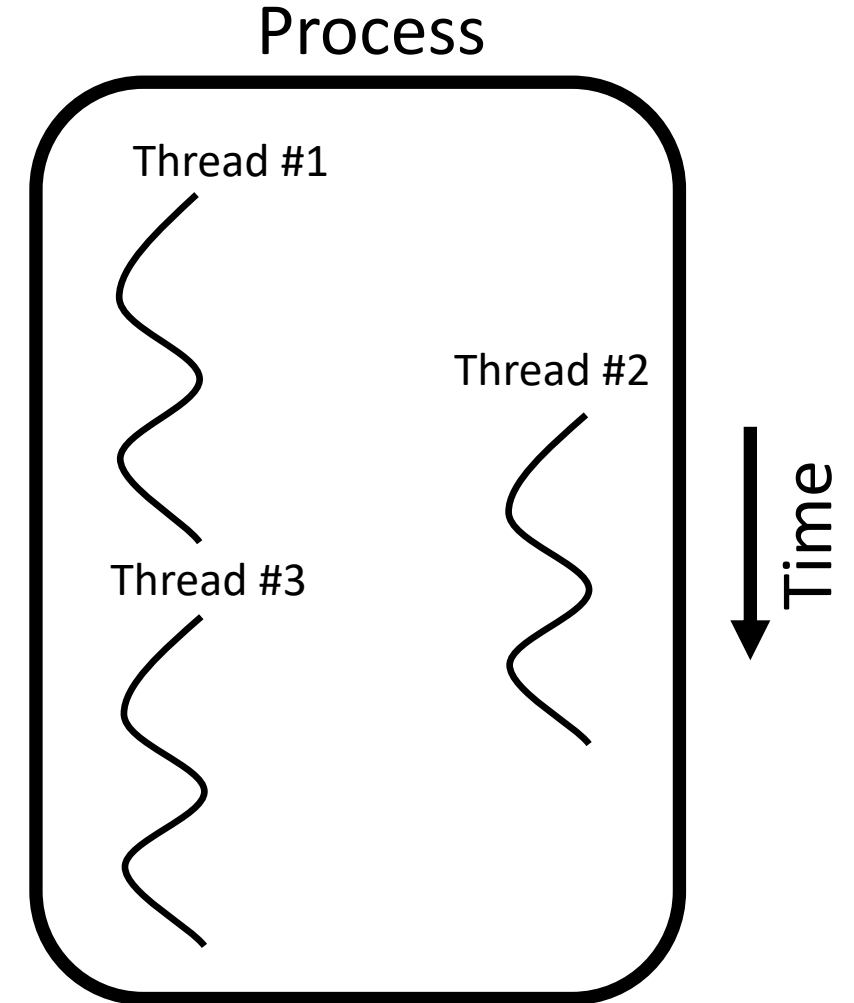


```
while (true) {  
    cout << "$ ";  
    cin >> command;  
    pid_t child = fork();  
    if (0 == child) {  
        execv(command, NULL);  
    } else {  
        wait(child);  
    }  
}
```



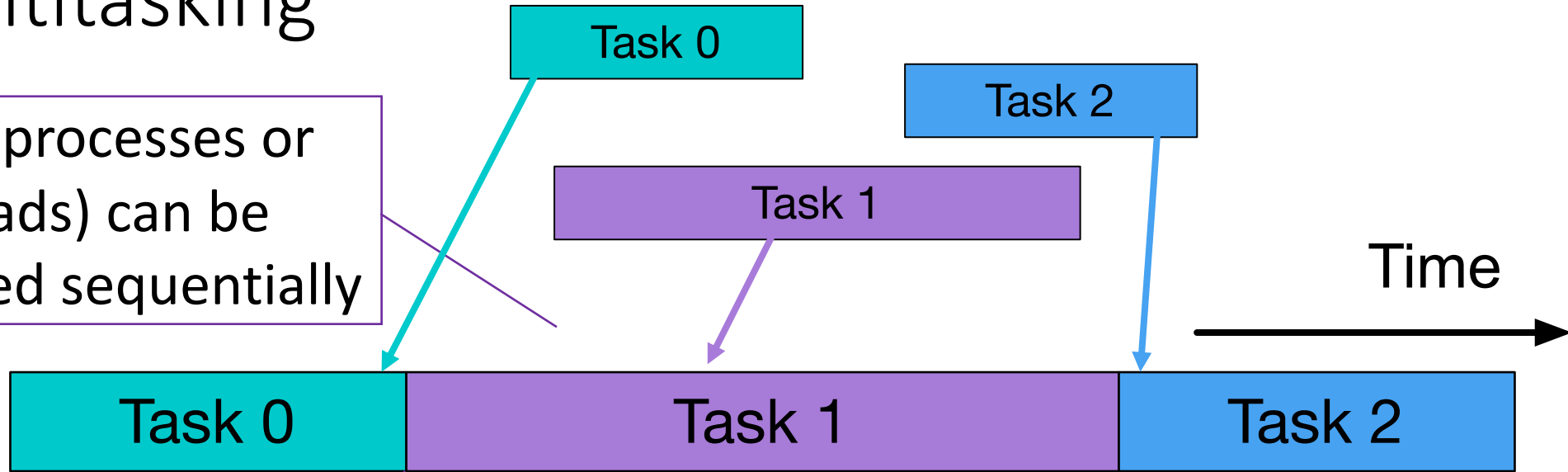
Processes and Threads

- A process is an abstraction for a collection of resources to represent a (running) program
 - CPU
 - Memory
 - Address space
- A thread is an abstraction of execution (using the resources within a process)
 - Can share an address space

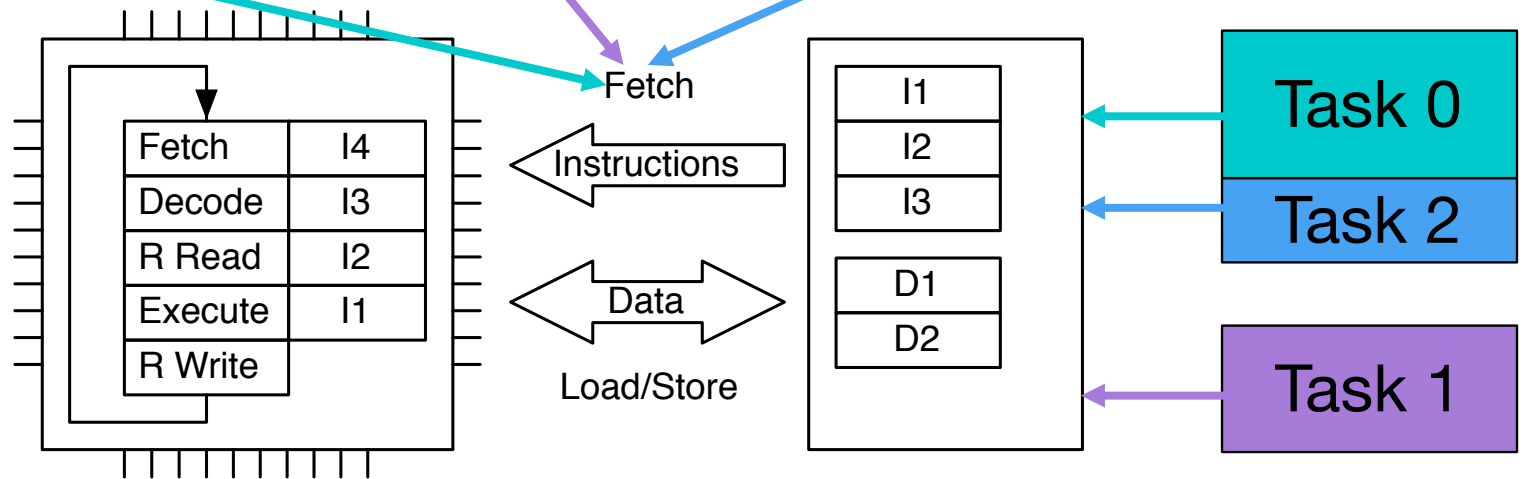
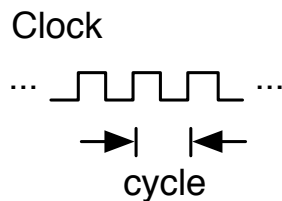


Multitasking

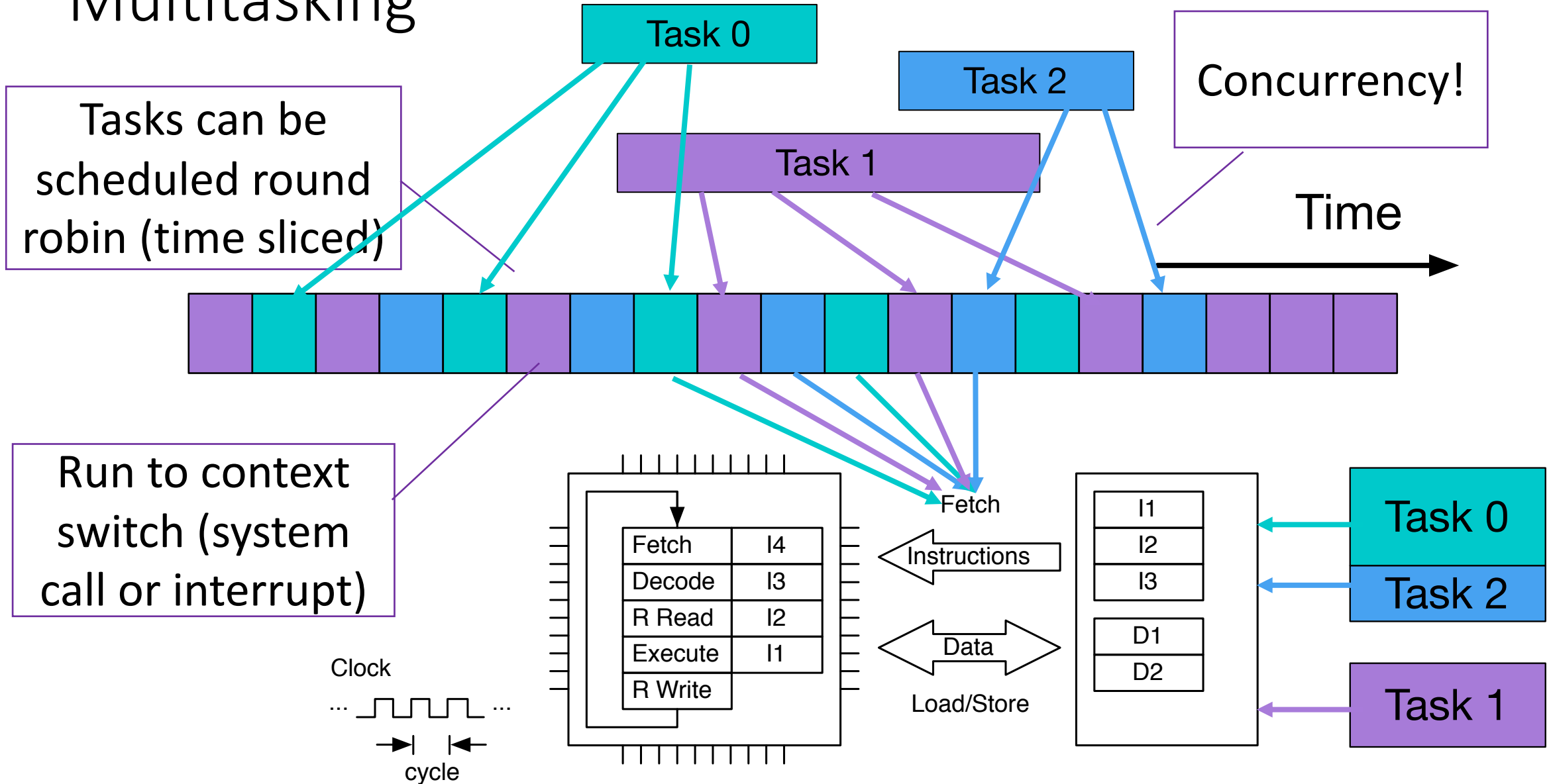
Tasks (processes or threads) can be scheduled sequentially



Run to completion



Multitasking



Multitasking on Multicore

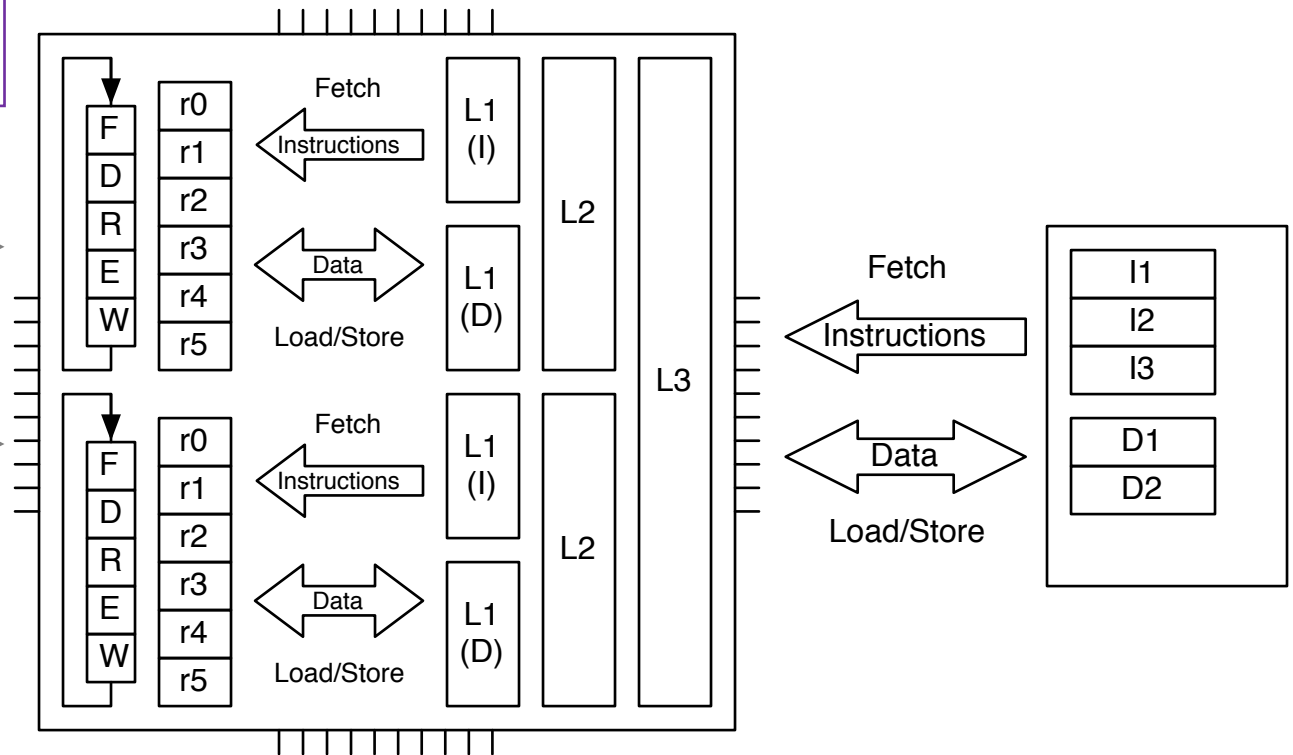
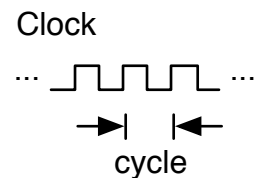
Concurrency!

Time sliced and mapped to separate cores

A single threaded task can only use one core at a time

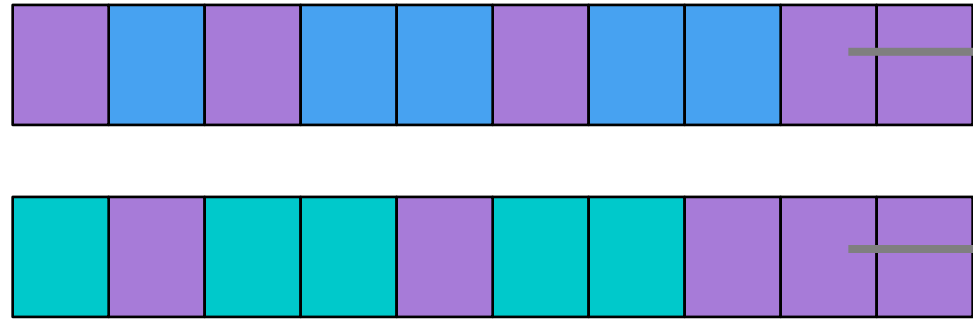


Time



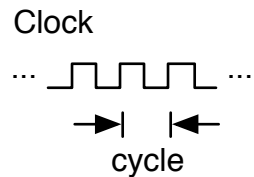
Multitasking on Multicore

Time sliced and mapped to separate cores



A multithreaded task can use multiple cores at a time

Time →



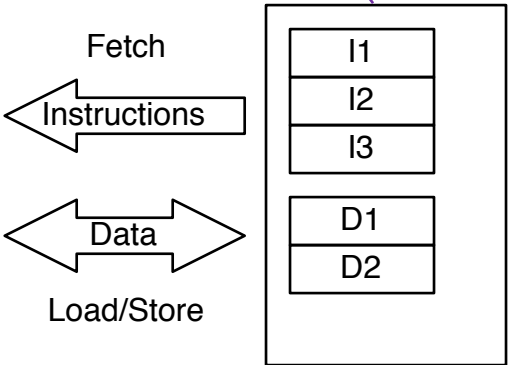
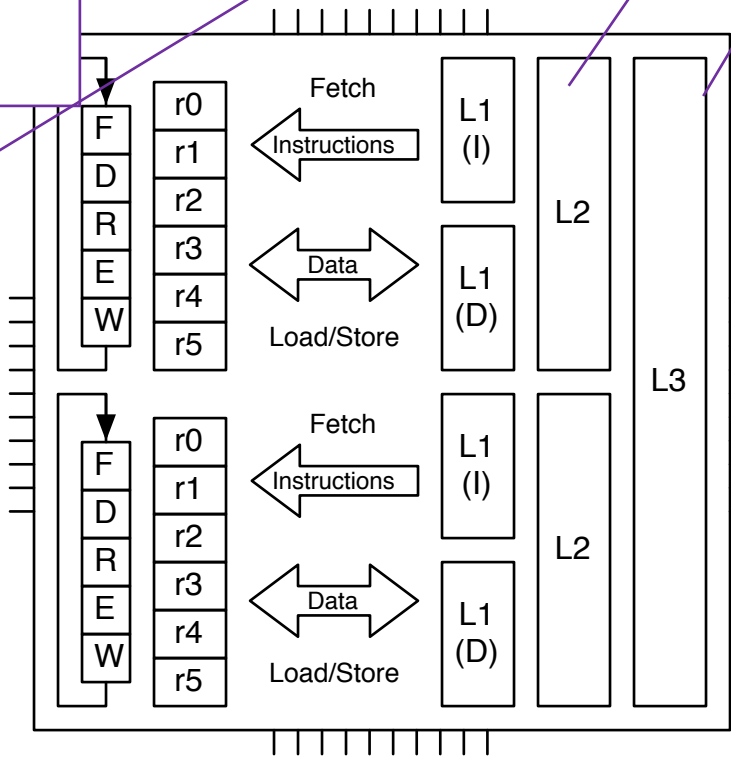
Parallelism!

Shorter run time!

What about L1, L2?

And L3 cache

Threads can share memory



Access same variables

Cache Coherence

Hardware managed

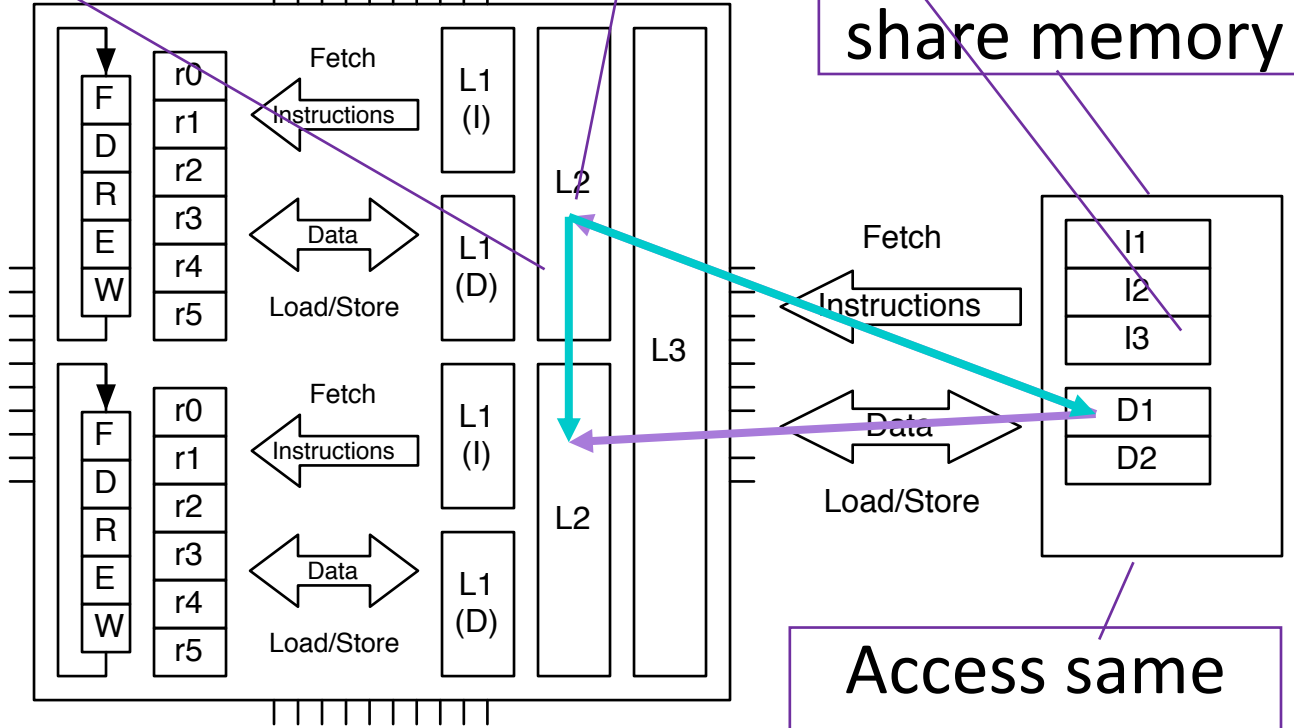
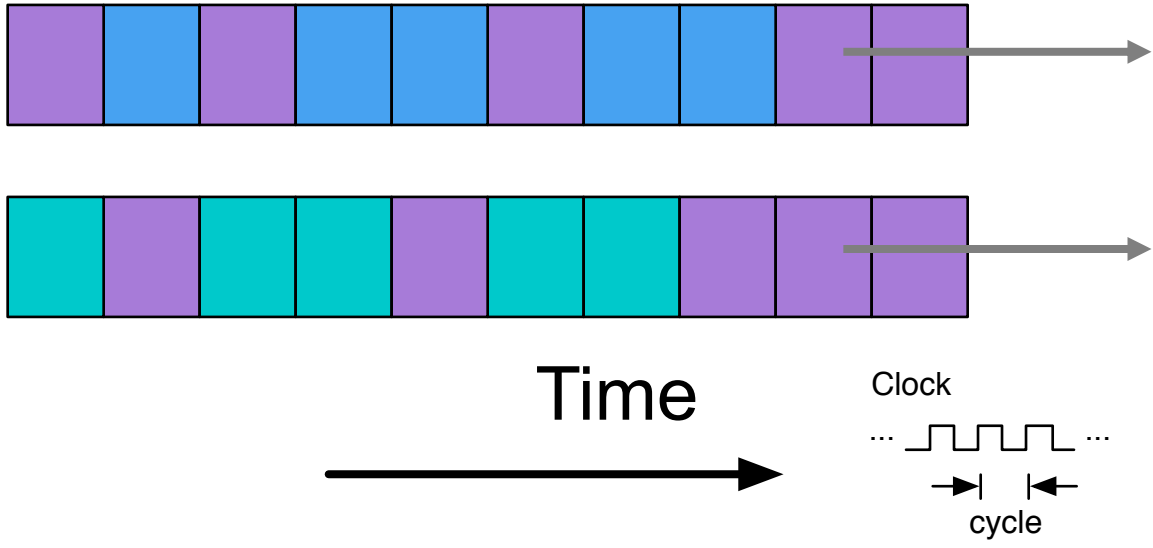
Same variable can be in two different caches

A multithreaded task can use multiple cores at a time

Cache coherence / memory consistency

What if one gets modified?

Threads can share memory



Access same variables

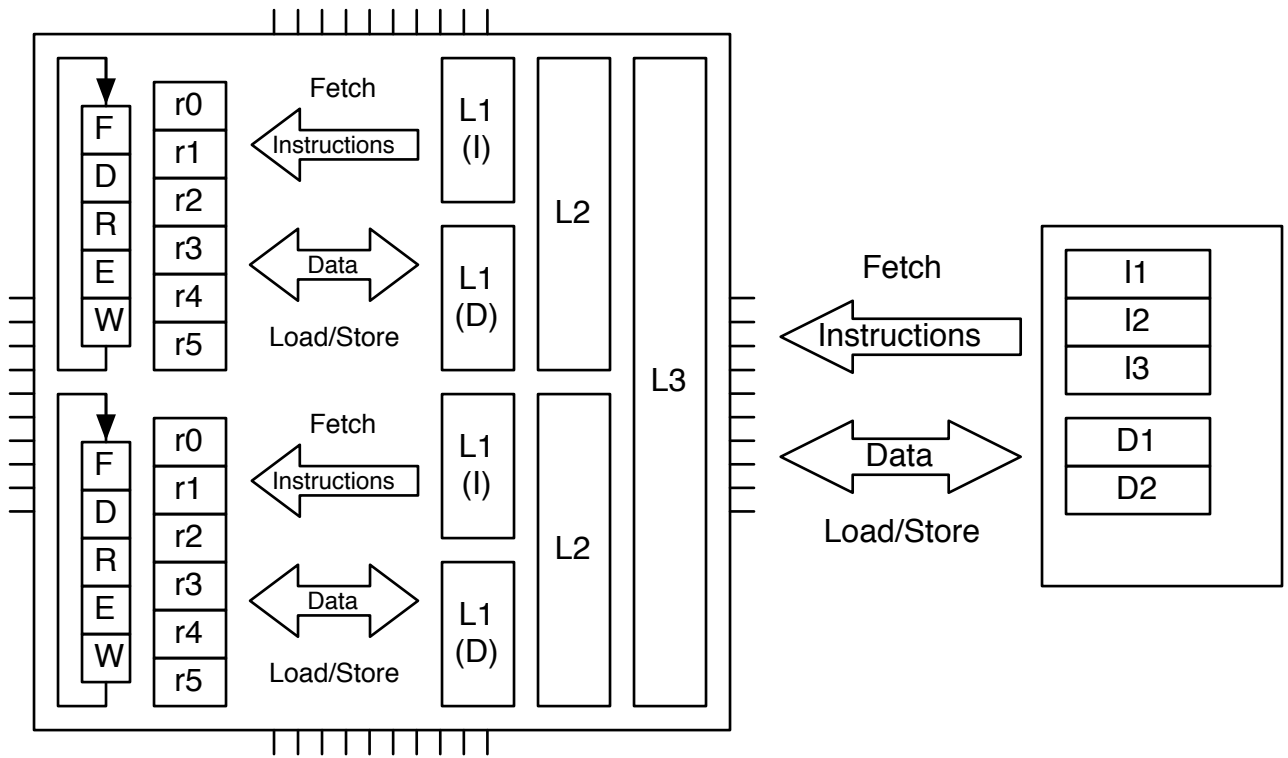
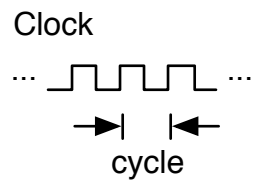
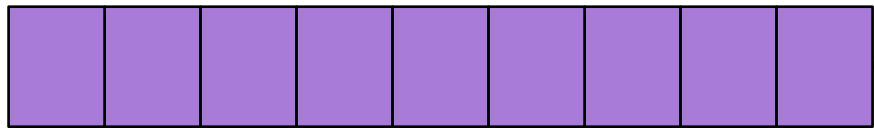
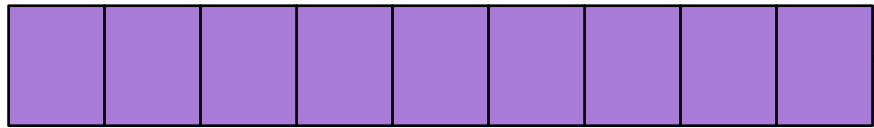
Multitasking on Multicore

Time



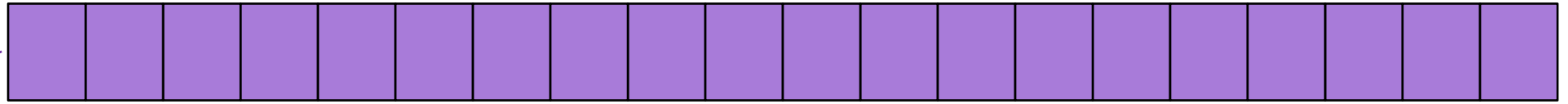
Run one task

In half the time (?)



Multitasking on Multicore

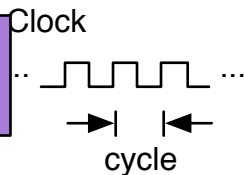
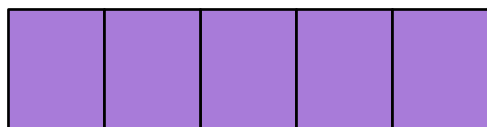
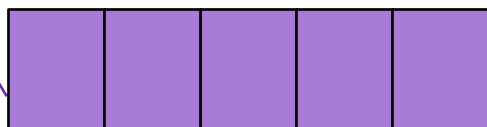
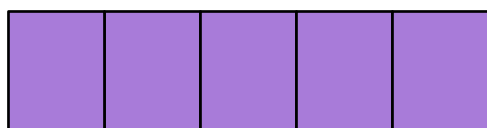
Time



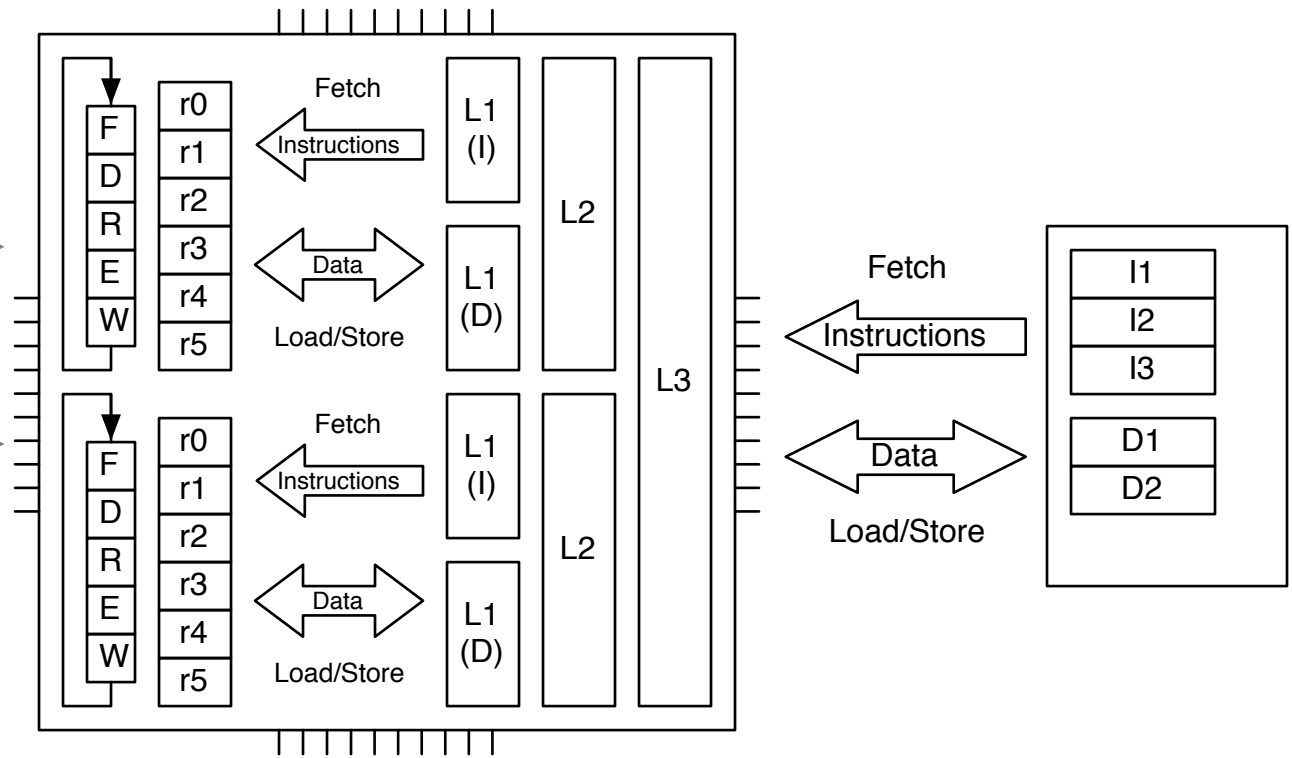
Run one task



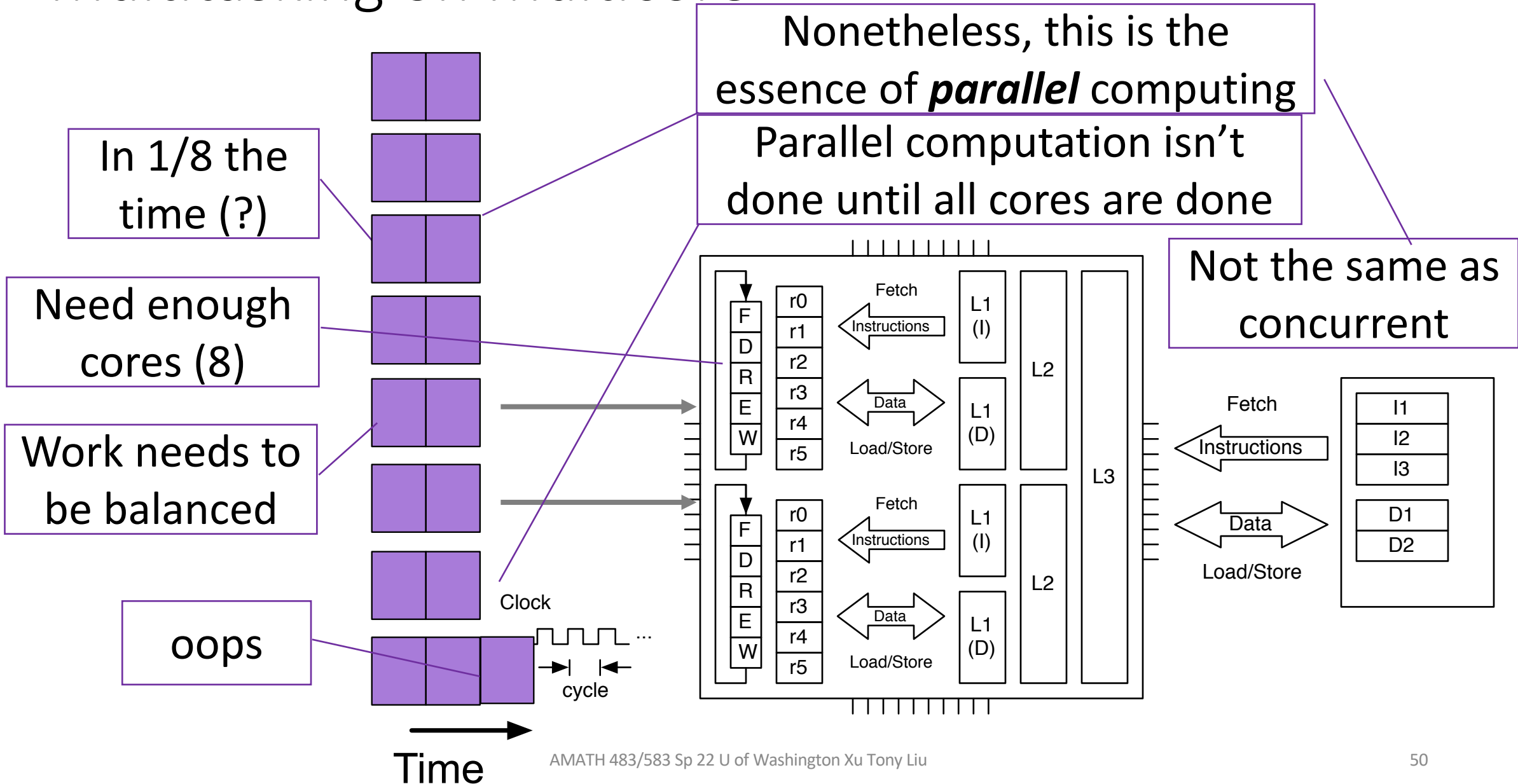
In 1/4 the time (?)



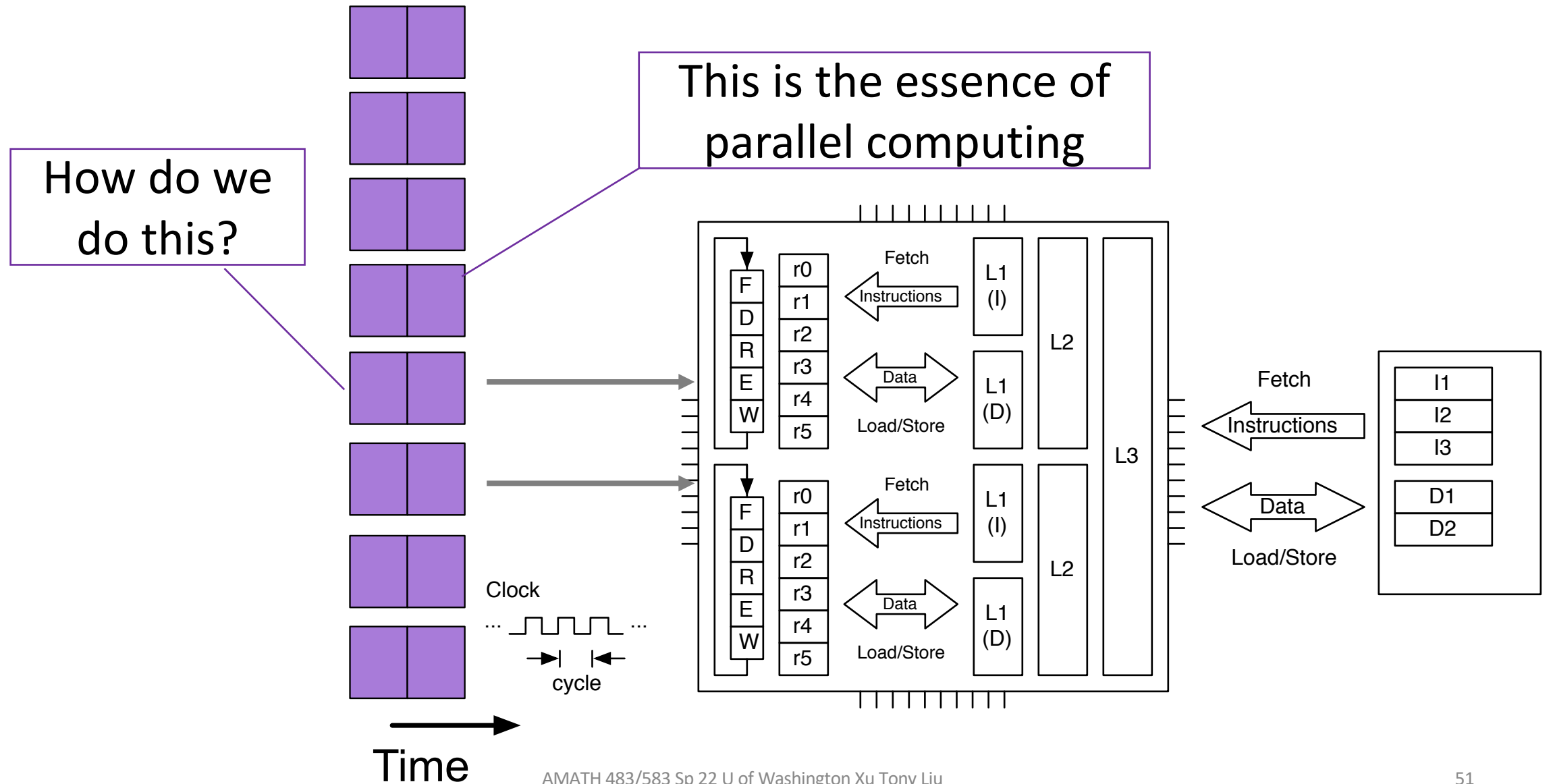
Time



Multitasking on Multicore

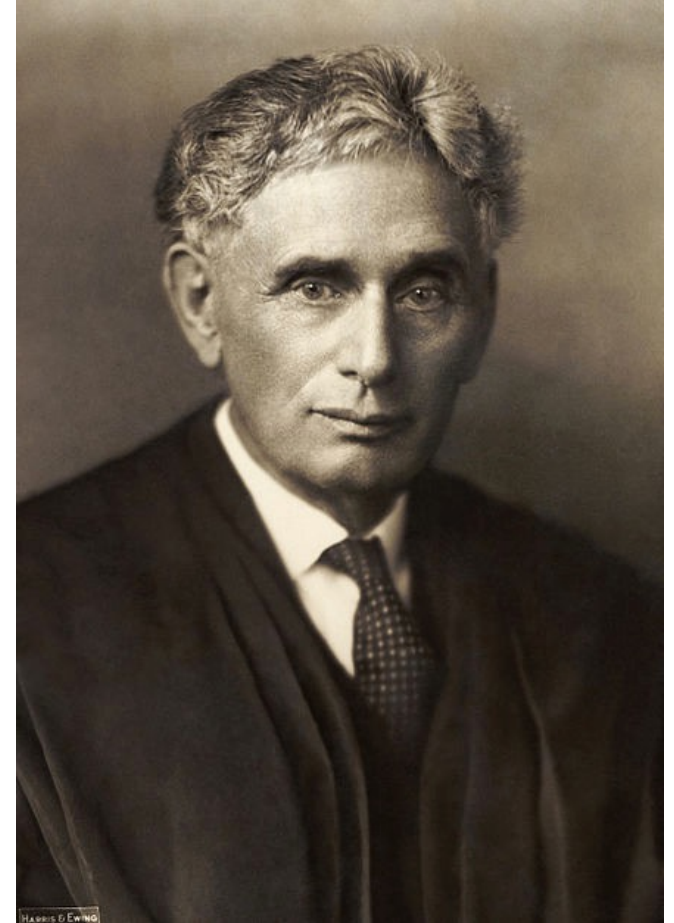


Multitasking on Multicore

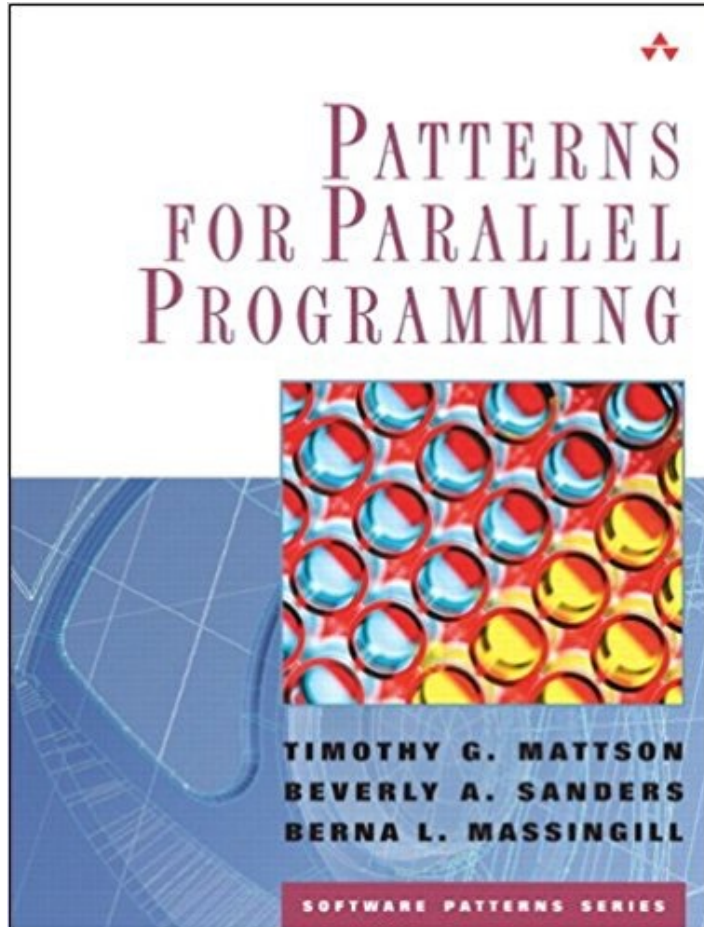


Processes vs Threads

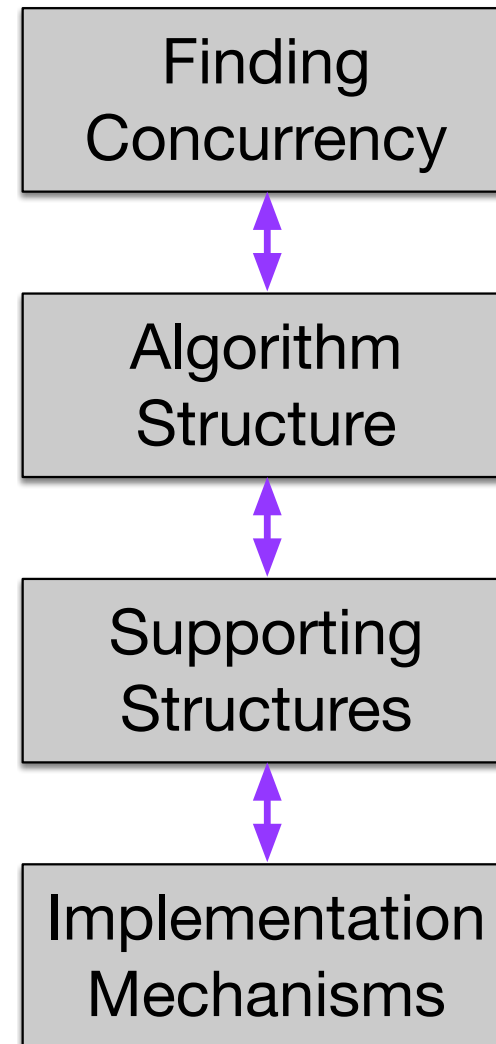
- A process is an abstraction for a collection of resources to represent a (running) program
 - Manage resources: CPU, Memory, Address space, etc.
 - Not light weighted
 - Expensive to create, and terminate
- A thread is an abstraction of execution (using the resources within a process)
 - The smallest sequence of programmed instructions
 - Threads can share an address space of a process
 - Light weighted
 - Less expensive to create and terminate



Parallelization Strategy



Timothy Mattson, Beverly Sanders, and Berna Massingill.
2004. *Patterns for Parallel Programming* (First ed.). Addison-
Wesley Professional.

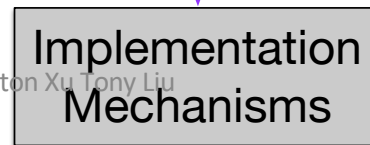
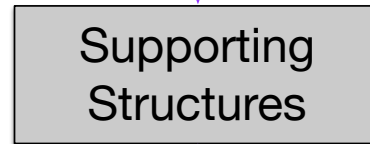
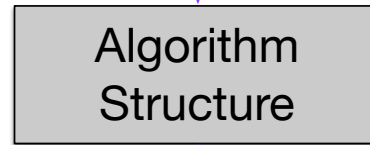
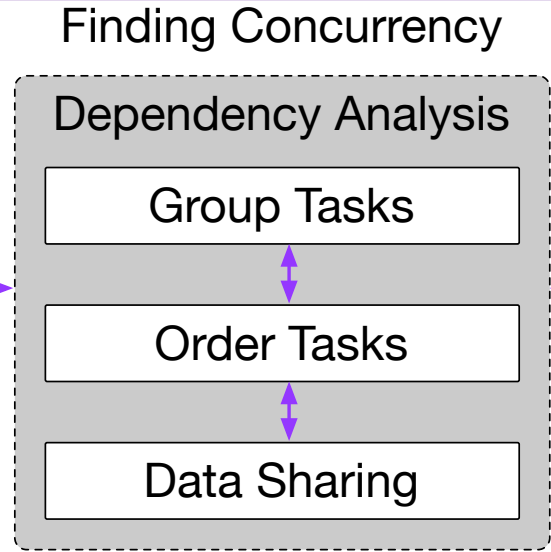
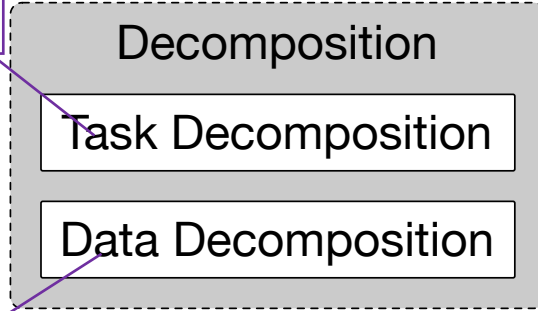


Finding Concurrency

Into tasks that can execute concurrently

Decompose problem into pieces that can execute concurrently

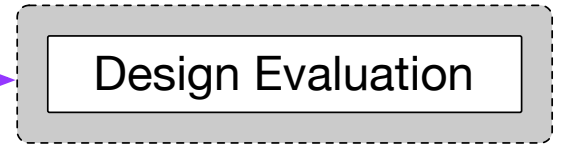
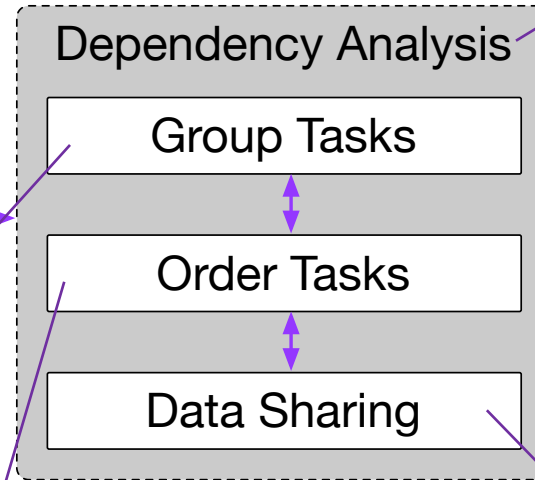
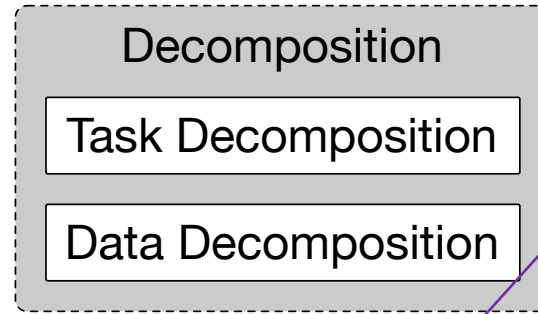
Units that can be operated on (relatively) independently



Finding Concurrency

Ways to group tasks to simplify management of dependencies

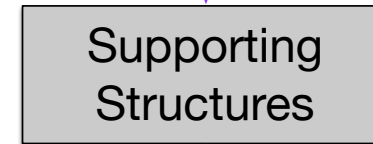
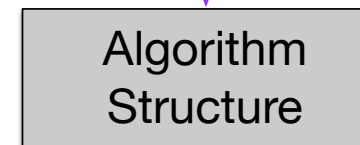
Finding Concurrency



Ways to group tasks to simplify management of dependencies

Ways to order tasks for correctness, other constraints

Given a decomposition, ways to share data among tasks



Algorithm Structure

Organize around concurrent tasks

Finding Concurrency

Fundamental organizing principle

Algorithm Structure

Organize by Tasks

Task Parallelism

Divide and Conquer

Organize by Data Decomposition

Geometric Decomposition

Recursive Data

Organize by Flow of Data

Pipeline

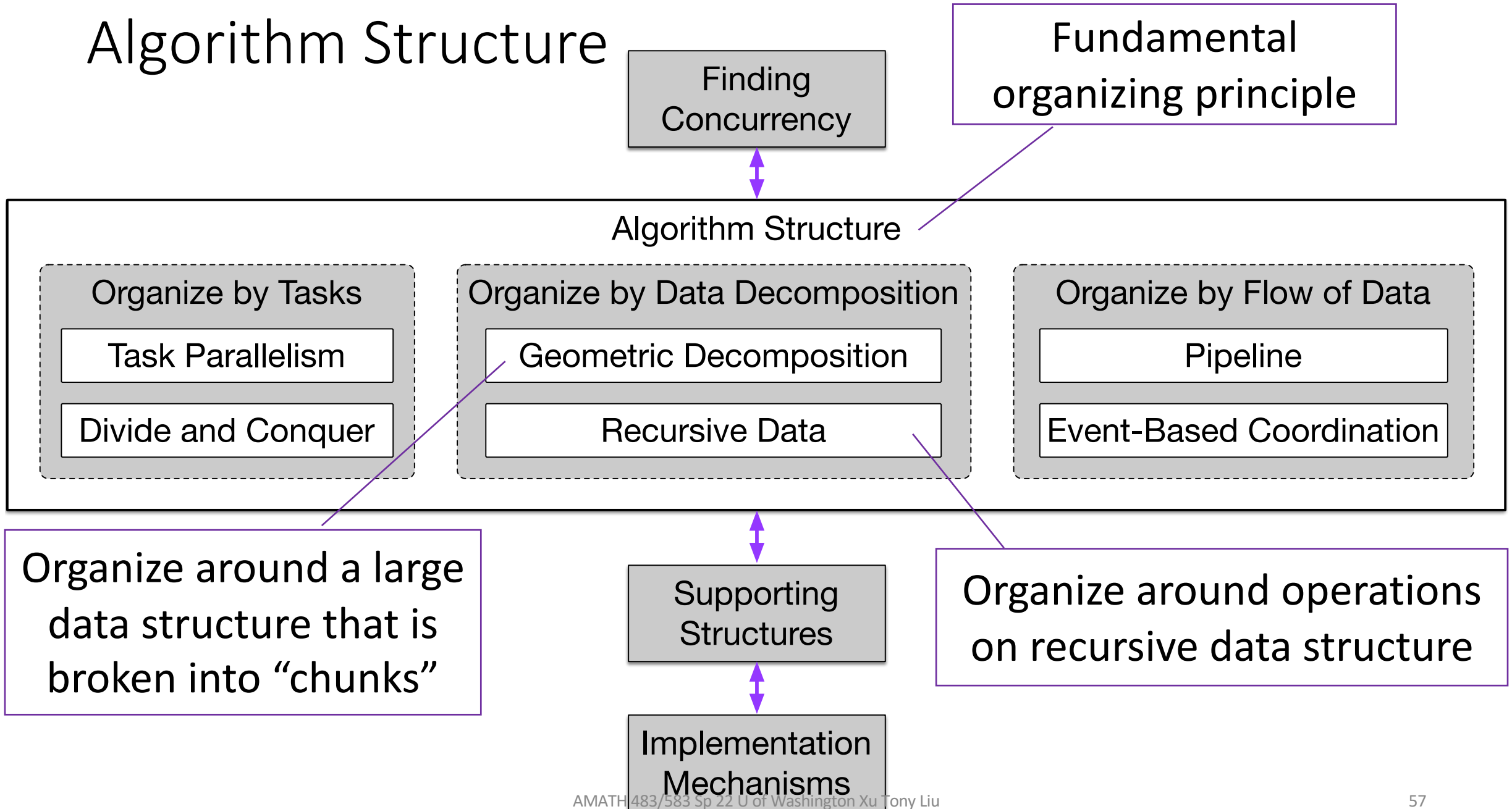
Event-Based Coordination

Exploit potential concurrency in divide and conquer algorithms

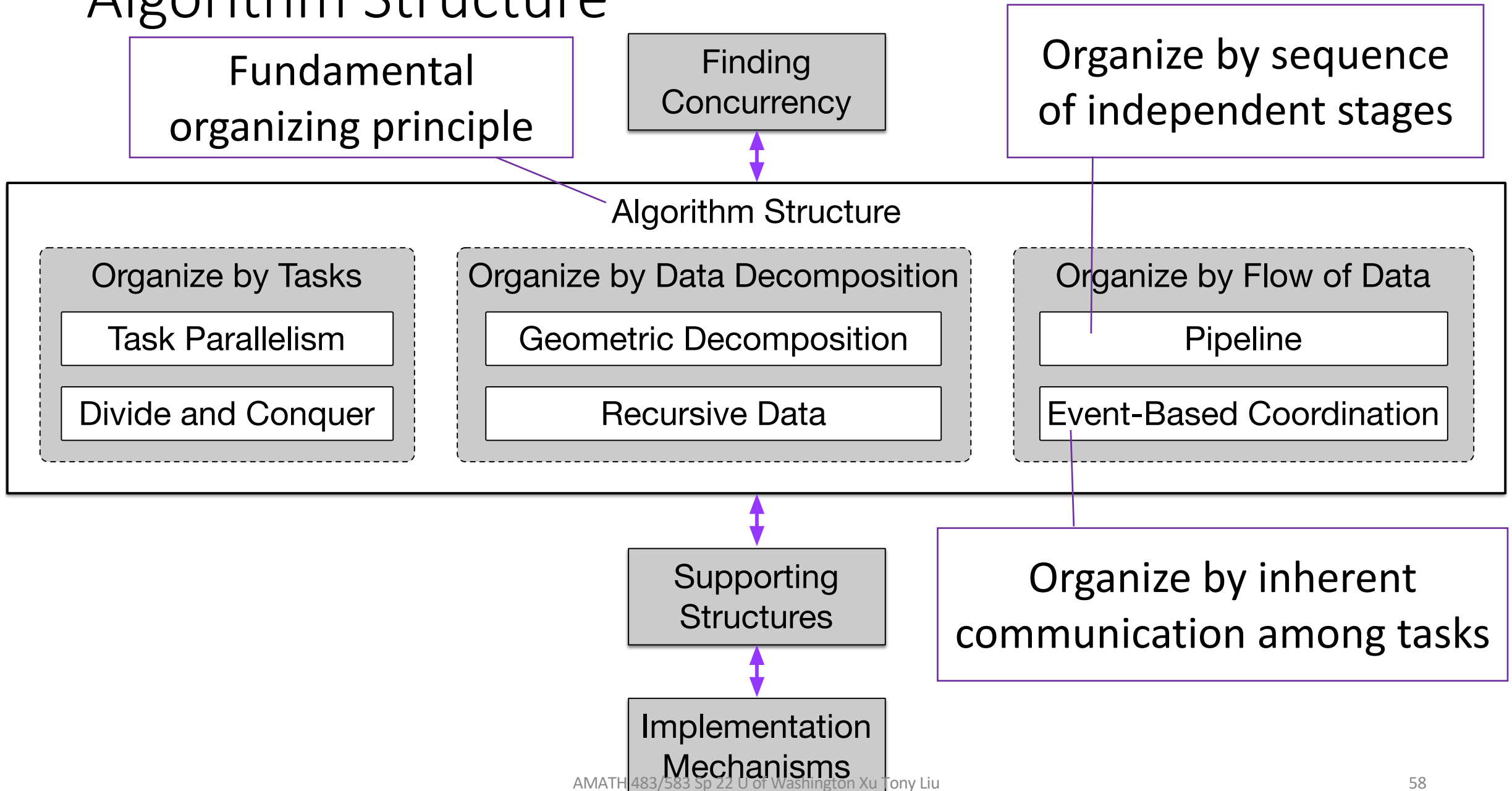
Supporting Structures

Implementation Mechanisms

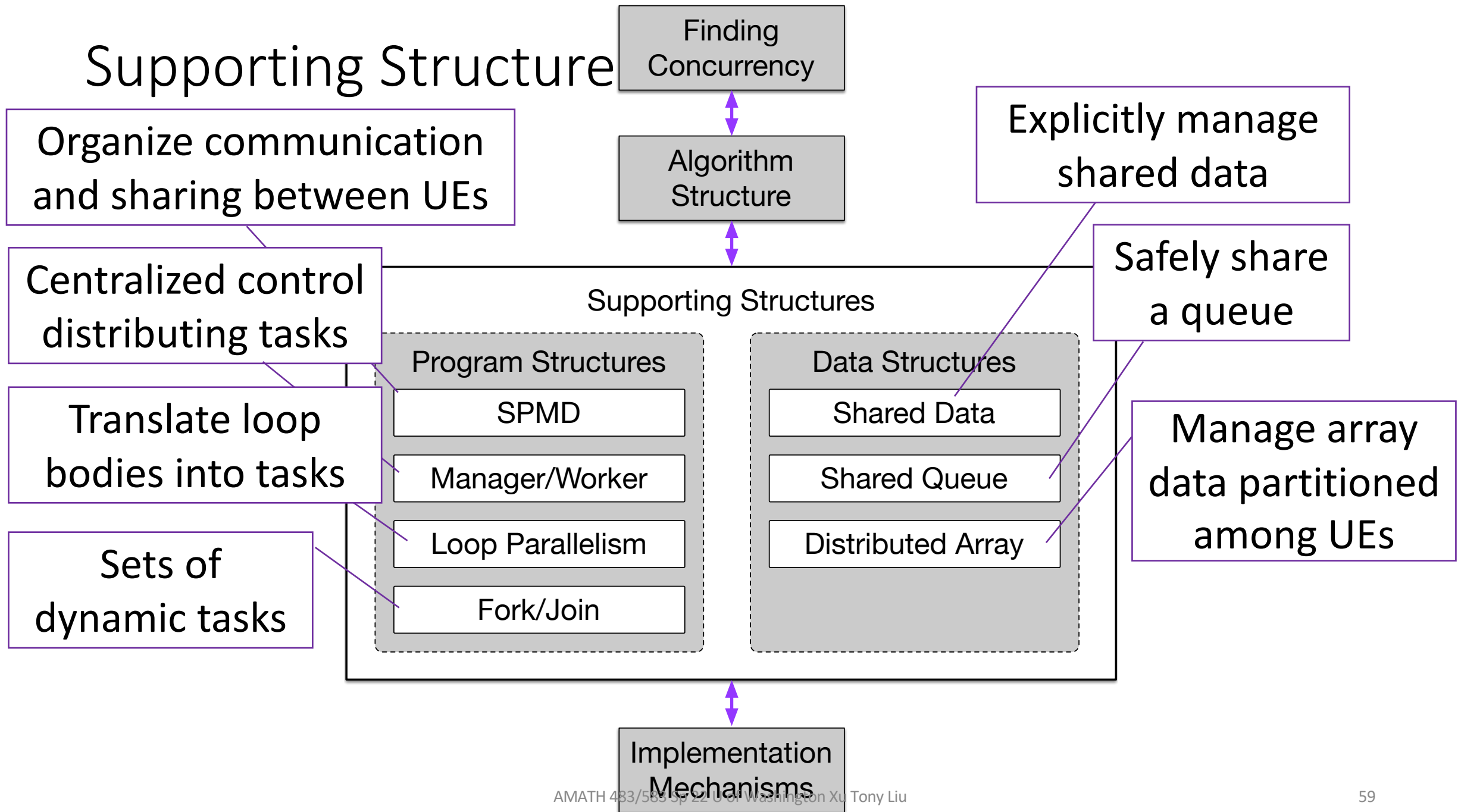
Algorithm Structure



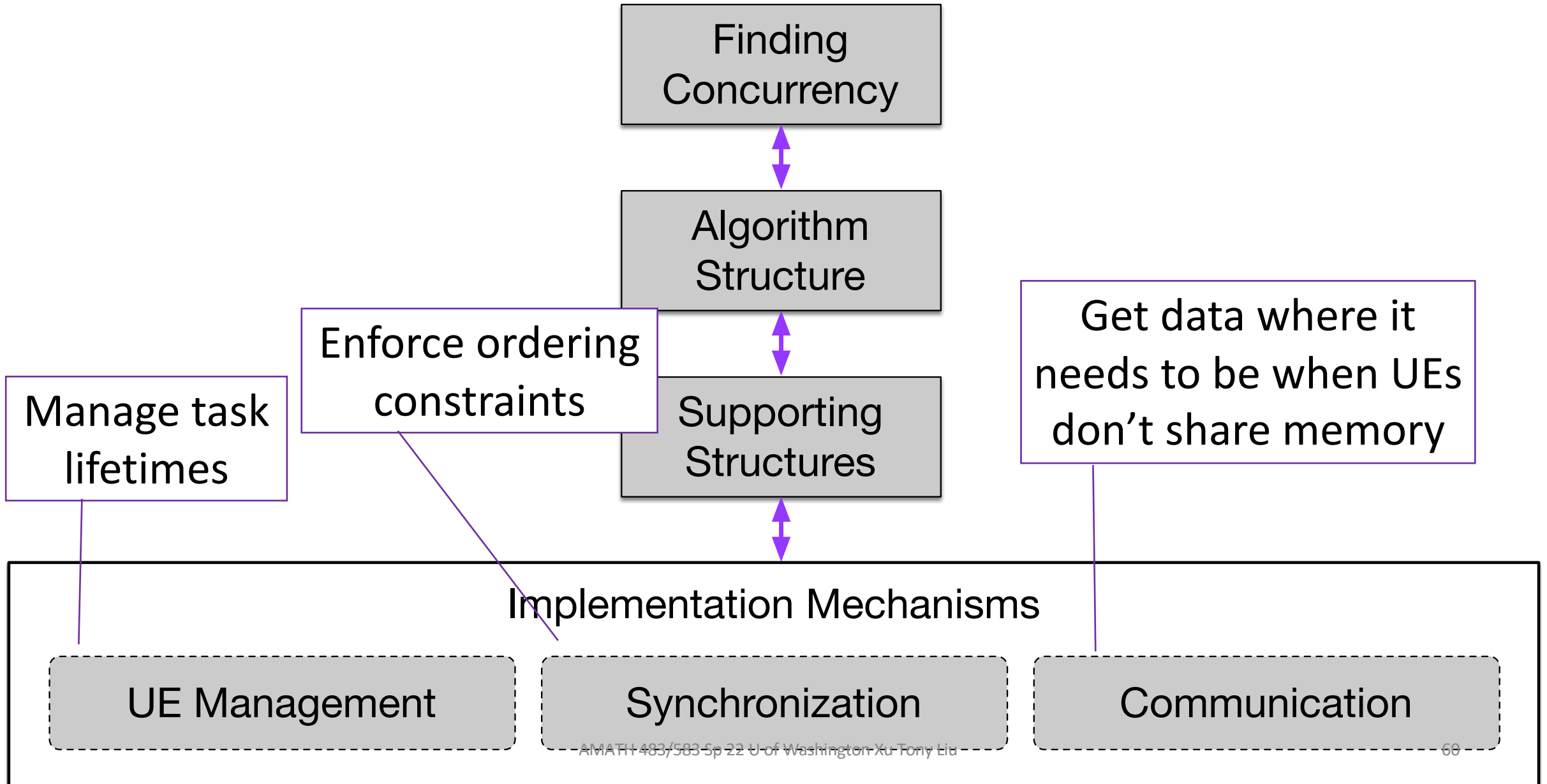
Algorithm Structure



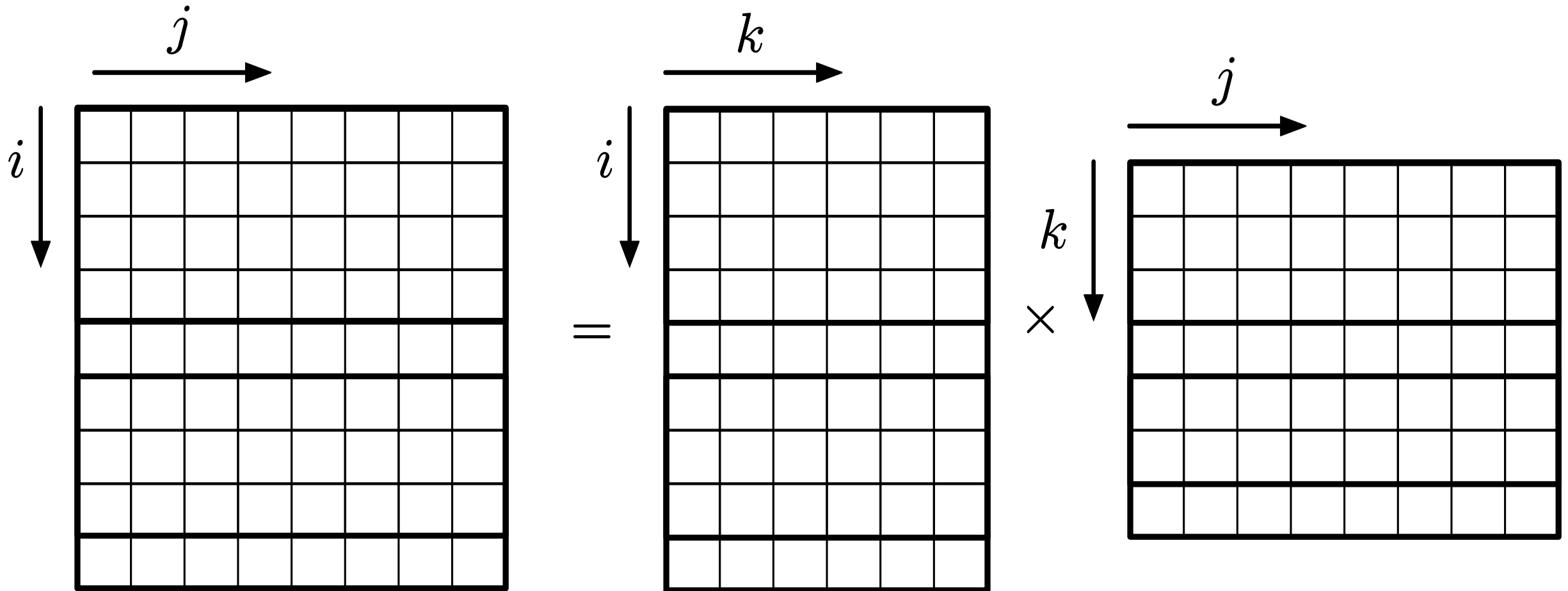
Supporting Structure



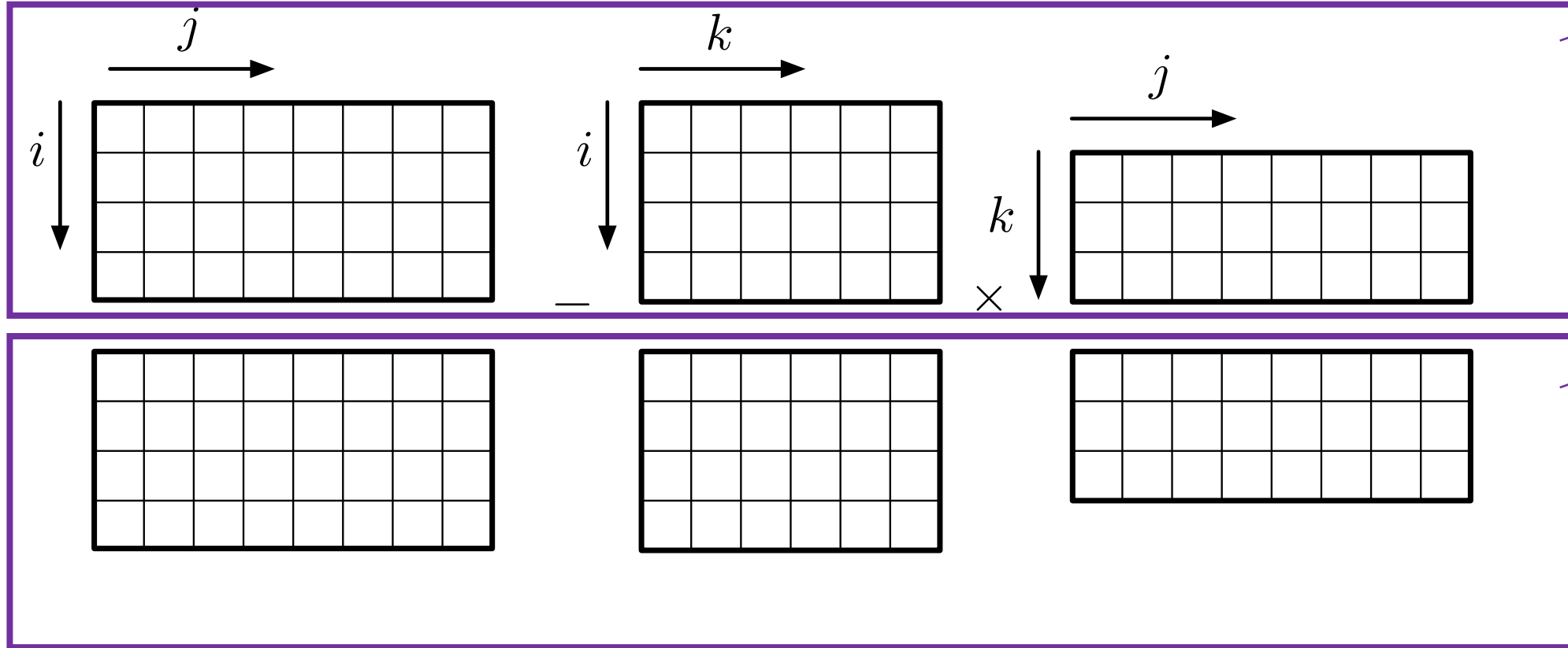
Implementation Mechanisms



Parallel Computing with Processes



Parallel Computing with Processes

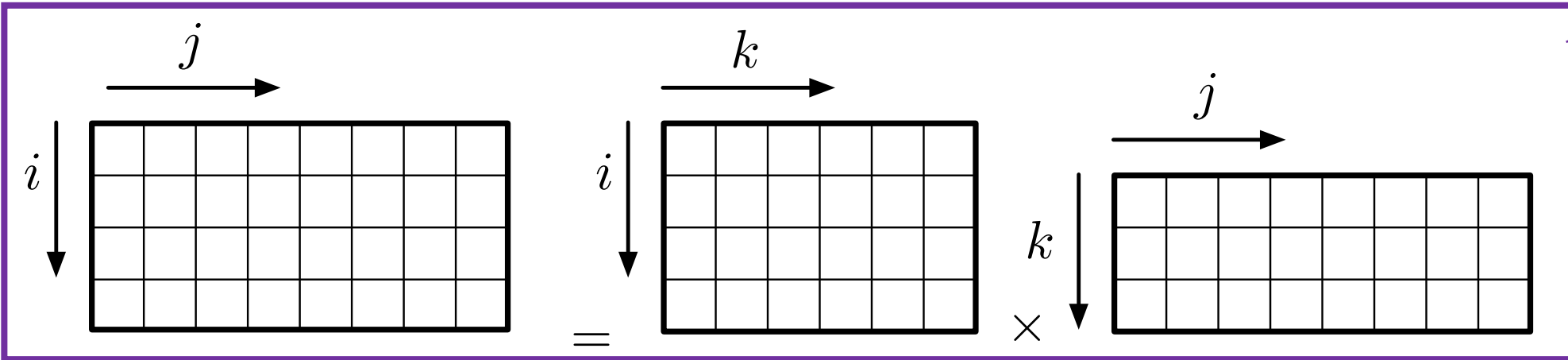


Process 0

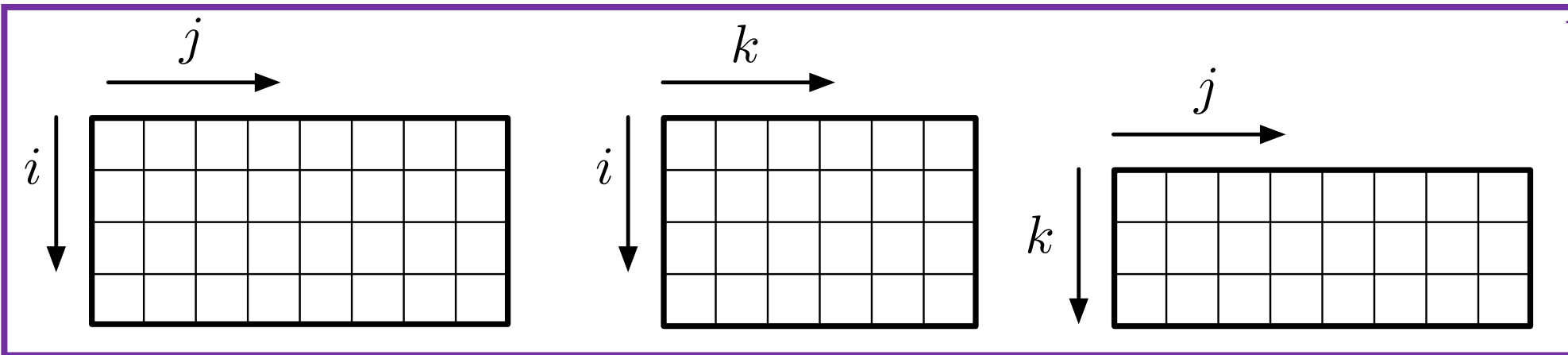
Process 1

Parallel Computing with Processes

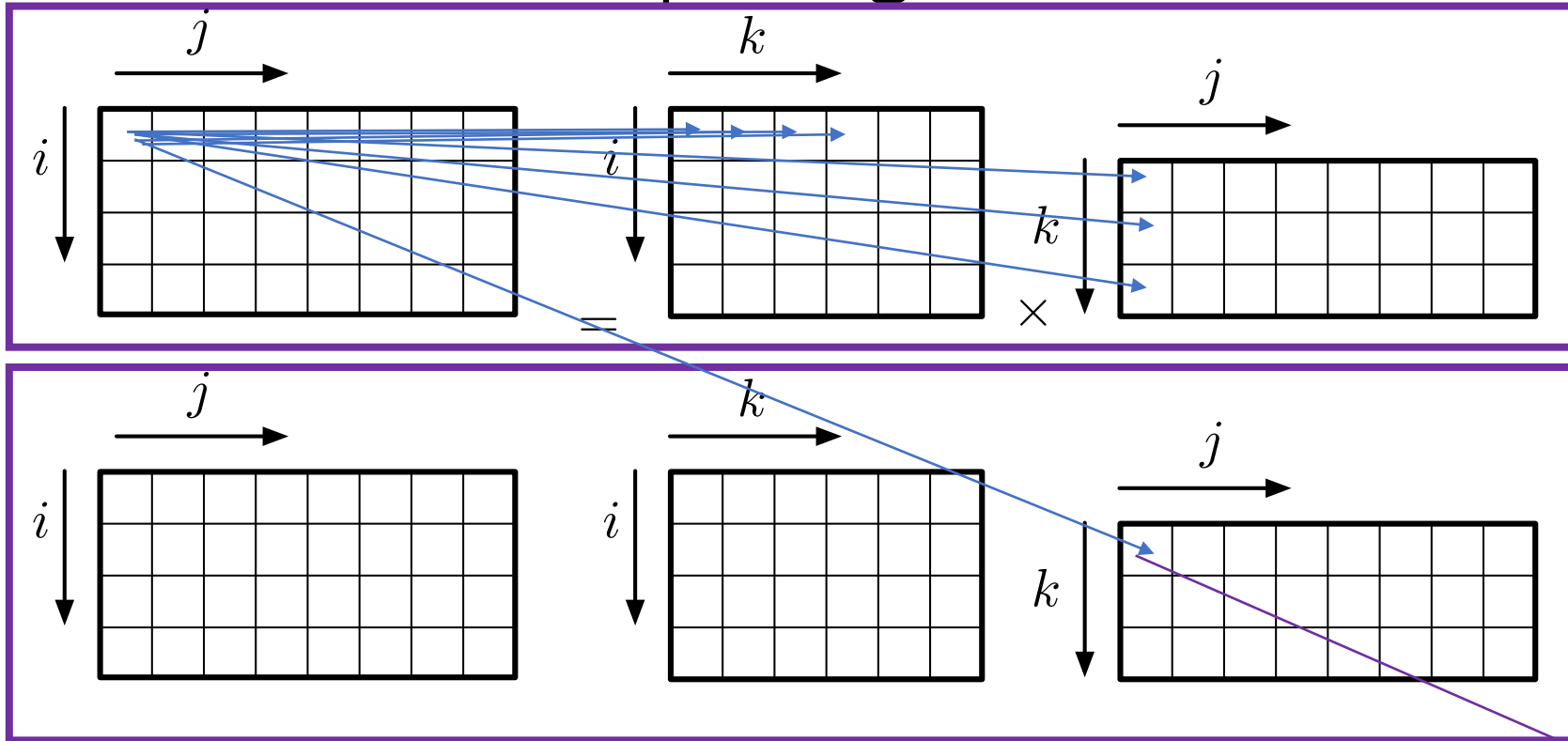
Process 0



Process 1



Parallel Computing with Processes



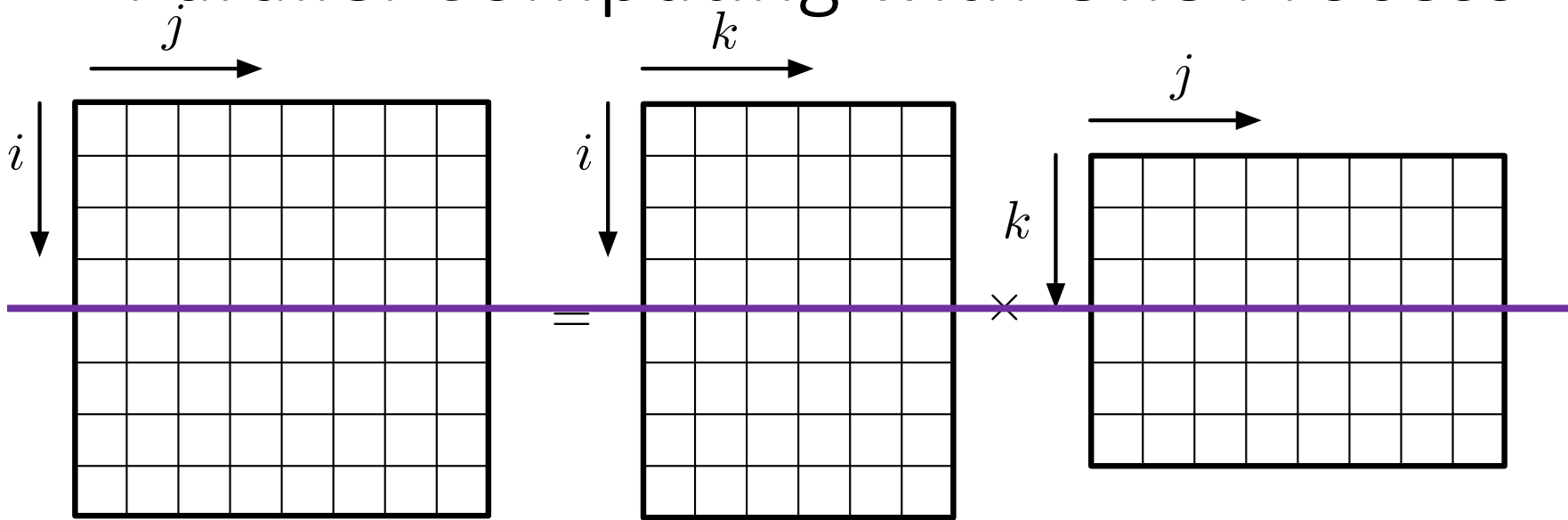
```
for (int i = 0; i < A.numRows(); ++i)
  for (int j = 0; j < B.numCols(); ++j)
    for (int k = 0; k < A.numCols(); ++k)
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

```
for (int i = 0; i < A.numRows(); ++i)
  for (int j = 0; j < B.numCols(); ++j)
    for (int k = 0; k < A.numCols(); ++k)
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Can't index from
different process b/c
different address space

```
for (int k = 0; k < A.numCols(); ++k) {
  C(i,j) += A(i,k) * B(k,j);
}
```

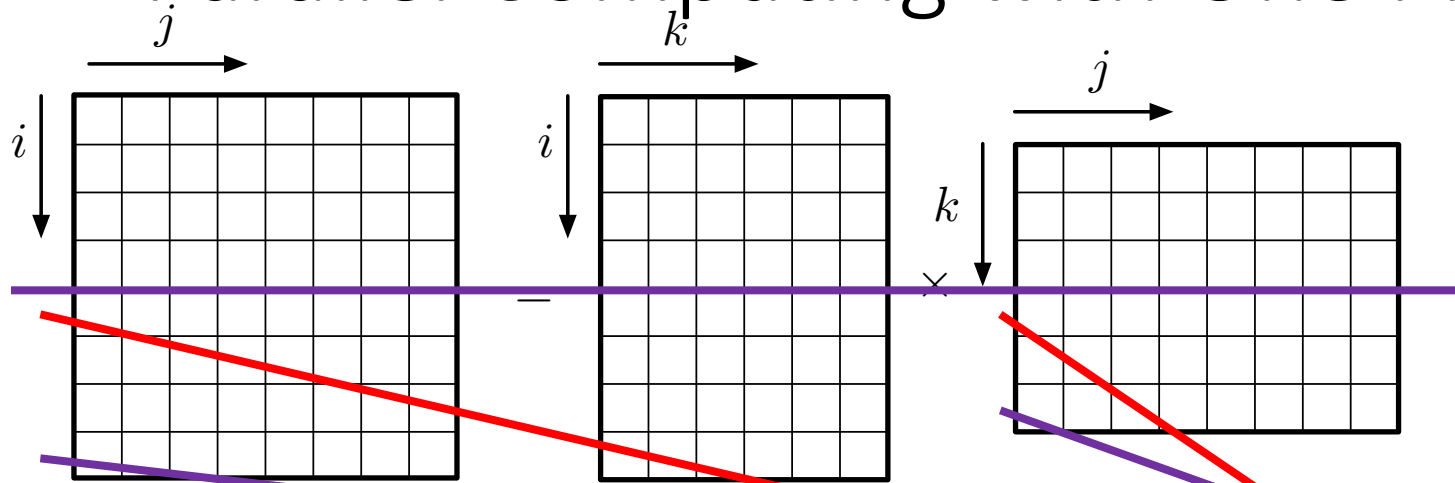

Parallel Computing with One Process



```
for (int i = 0; i < A.numRows(); ++i) {  
    for (int j = 0; j < B.numCols(); ++j) {  
        for (int k = 0; k < A.numCols(); ++k) {  
            C(i,j) += A(i,k) * B(k,j);  
        }  
    }  
}
```

```
for (int i = 0; i < A.numRows(); ++i) {  
    for (int j = 0; j < B.numCols(); ++j) {  
        for (int k = 0; k < A.numCols(); ++k) {  
            C(i,j) += A(i,k) * B(k,j);  
        }  
    }  
}
```

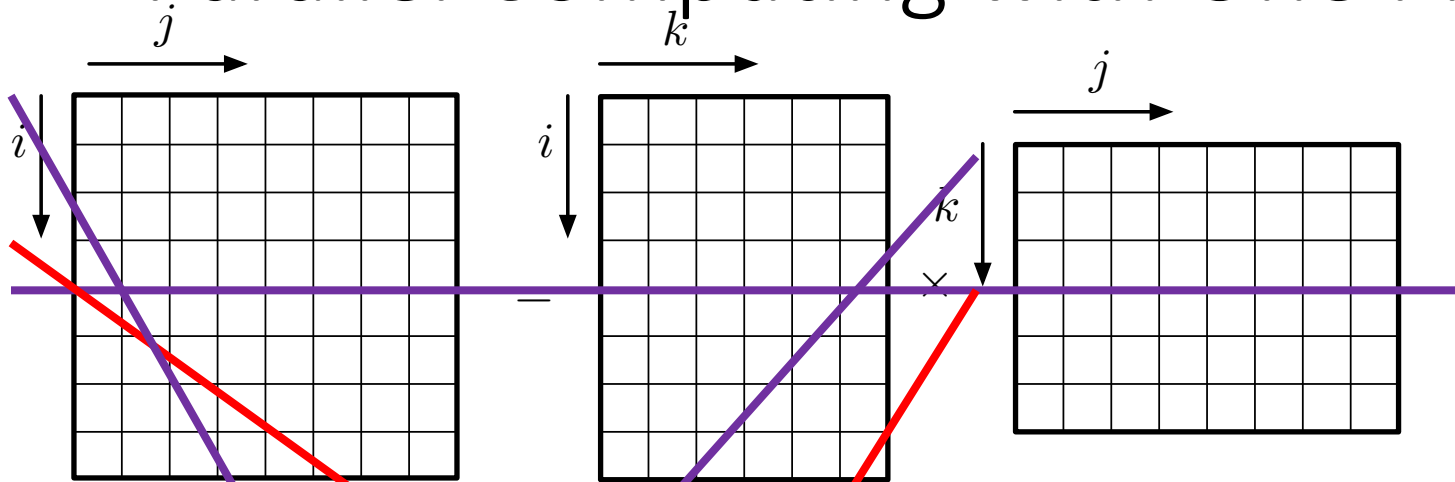

Parallel Computing with One Process



```
for (int i = 0; i < A.numRows(); ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols(); ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

```
for (int i = 0; i < A.numRows(); ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols(); ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

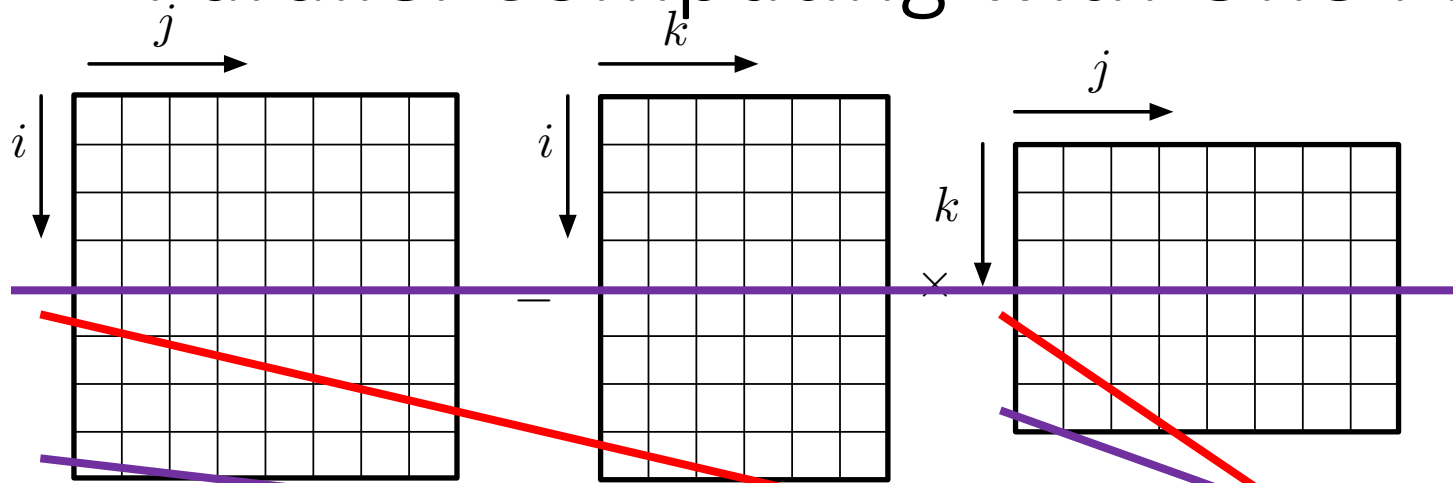
Parallel Computing with One Process



```
for (int i = 0; i < A.numRows(); ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols(); ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

```
for (int i = 0; i < A.numRows(); ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols(); ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

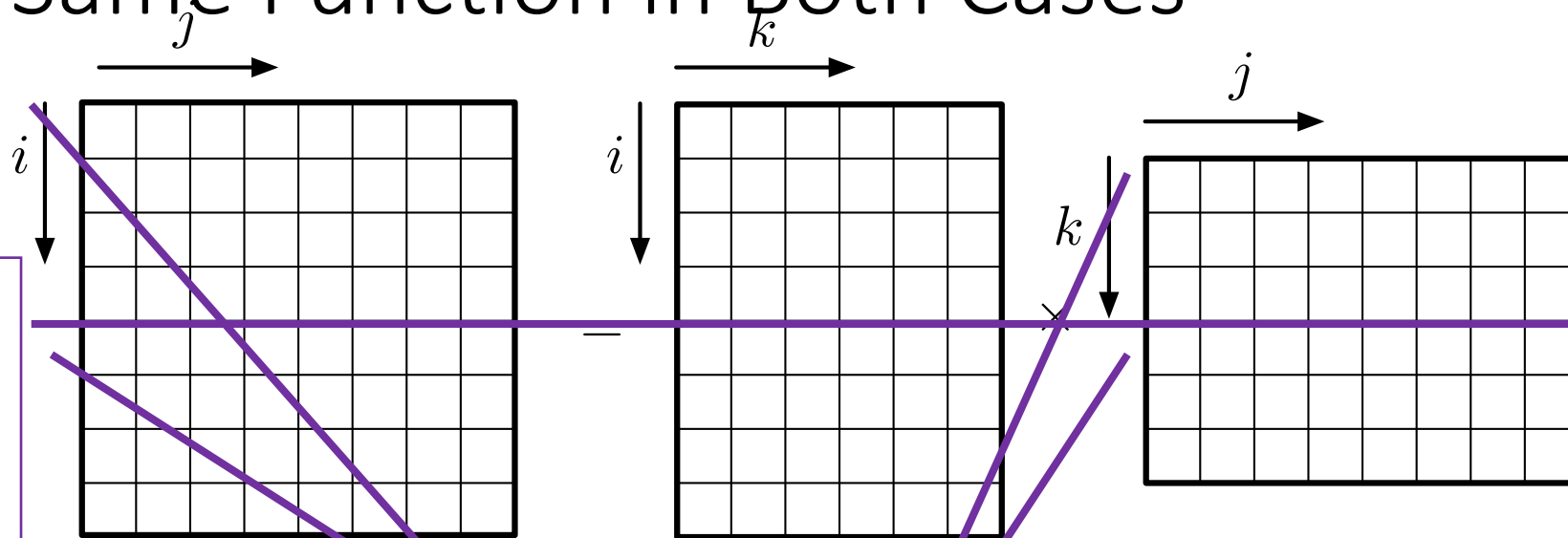
Parallel Computing with One Process



```
for (int i = 0; i < A.numRows()/2; ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols()/2; ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

```
for (int i = 0; i < A.numRows(); ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = 0; k < A.numCols(); ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

Use Same Function in Both Cases

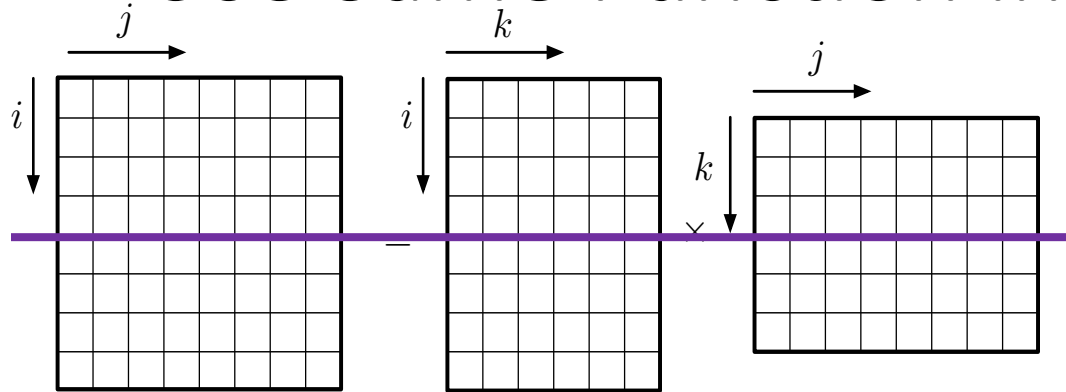


Still need to run two separate instances

Need to run them in **parallel** to get improved performance

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
  for (int j = 0; j < B.numCols(); ++j) {
    for (int k = kStart; k < kStart + A.numCols()/; ++k) {
      C(i,j) += A(i,k) * B(k,j);
    }
  }
}
```

Use Same Function in Both Cases



Run this

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

`int iStart = 0;`
`int kStart = 0;`

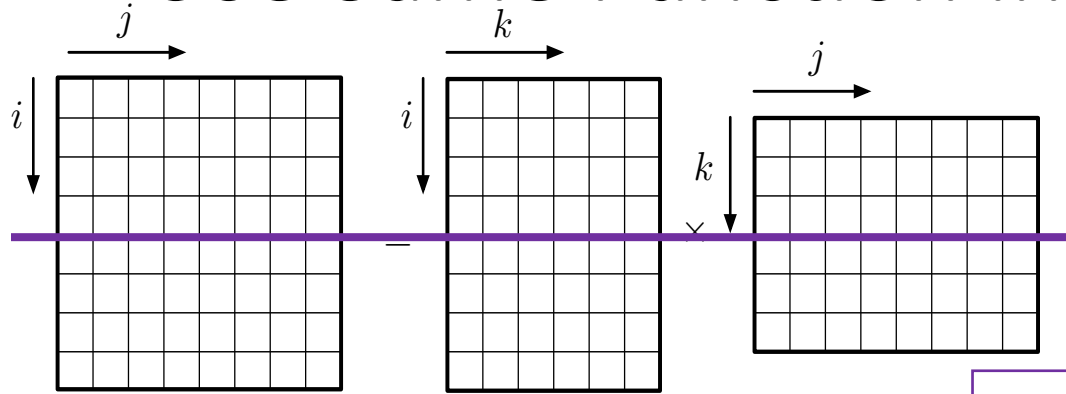
```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {  
  for (int j = 0; j < B.numCols(); ++j) {  
    for (int k = kStart; k < kStart + A.numCols()/2; ++k) {  
      C(i,j) += A(i,k) * B(k,j);  
    }  
  }  
}
```

`int iStart = A.numRows()/2;`
`int kStart = A.numCols()/2;`

Improved performance?

Then this

Use Same Function in Both Cases



At the same time

2X faster (?)

Run this

And this

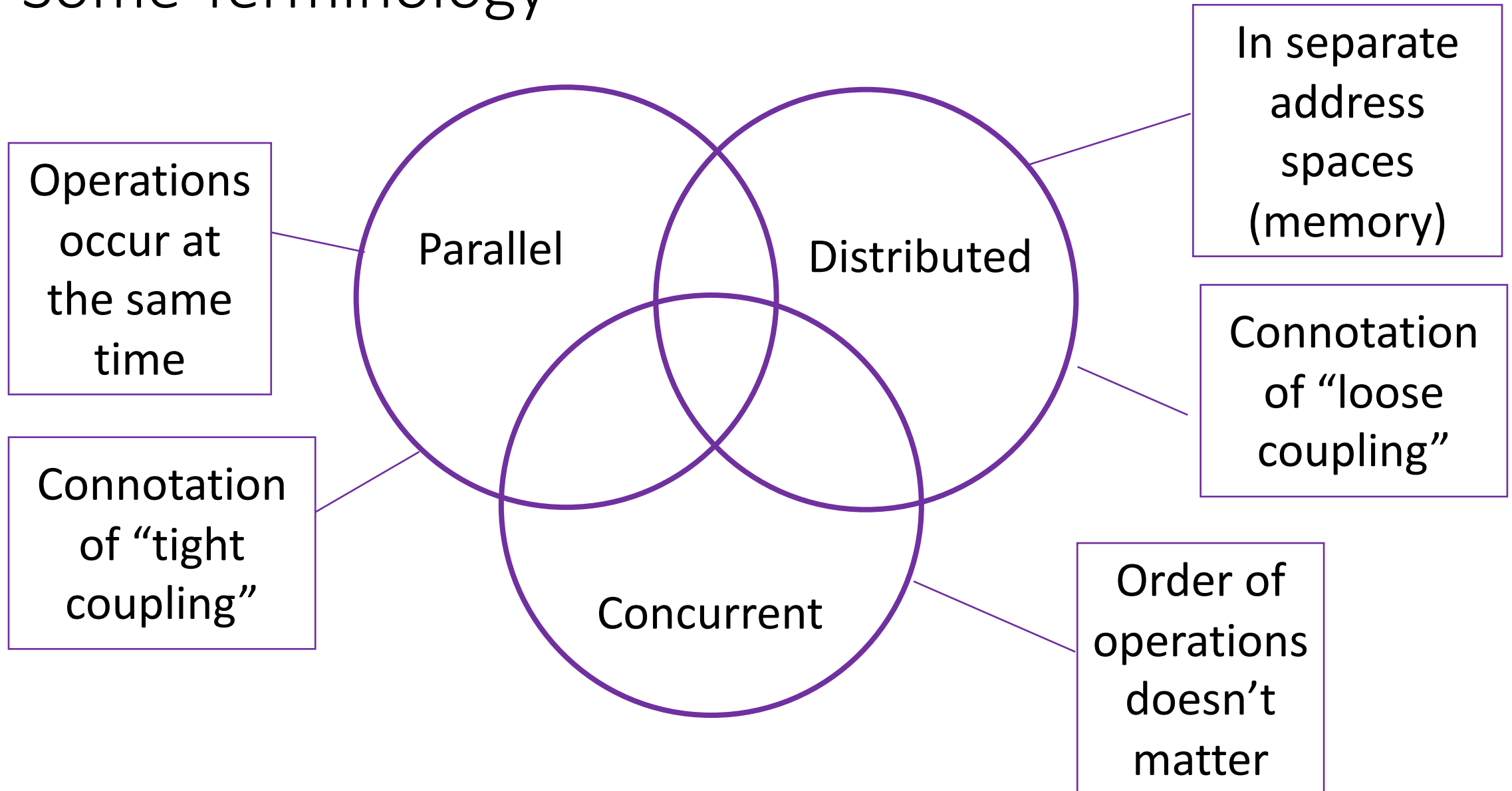
```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
    for (int j = 0; j < B.numCols(); ++j) {
        for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
            C(i,j) += A(i,k) * B(k,j);
        }
    }
}
```

int iStart = 0;
int kStart = 0;

```
for (int i = iStart; i < iStart + A.numRows()/2; ++i) {
    for (int j = 0; j < B.numCols(); ++j) {
        for (int k = kStart; k < kStart + A.numCols()/2; ++k) {
            C(i,j) += A(i,k) * B(k,j);
        }
    }
}
```

int iStart = A.numRows()/2;
int kStart = A.numCols()/2;

Some Terminology



Running Things “At the Same Time” vs Running Things At the Same Time

- Historically, threads evolved as a concurrency mechanism, not parallelism
- Enabled OS and processes to do multiple things “at the same time”
 - Running things “at the same time” – concurrency
- Can be used for performance if threads are executed in parallel
 - Running things at the same time (literally) - parallelism
 - Parallelism is the task of running one computation (part of it) simultaneously

Running Things At the Same Time in C++

```
#include <iostream>
#include <thread>
using namespace std;
```

Pull in thread library

```
void sayHello() {
    cout << "Hello World" << endl;
}
```

Simple function

Create a thread

```
int main() {
    thread helloThread(sayHello);
    helloThread.join();
    return 0;
}
```

Join back to main thread

That runs this function

Multithreading

```
void sayHello(int tnum) {
    cout << "Hello World.  I am thread " << tnum << endl;
}

int main() {
    std::thread tid[16];

    for (int i = 0; i < 16; ++i)
        tid[i] = thread (sayHello, i);

    for (int i = 0; i < 16; ++i)
        tid[i].join();

    return 0;
}
```

Multithreading

```
void sayHello(int tnum) {  
    cout << "Hello World. I am thread " << tnum << endl;  
}  
  
int main() {  
    std::thread tid[16];  
  
    for (int i = 0; i < 16; ++i)  
        tid[i] = thread (sayHello, i);  
  
    for (int i = 0; i < 16; ++i)  
        tid[i].join();  
  
    return 0;  
}
```

Program
output

\$./a.out

Hello World. I am thread Hello World. I am thread Hello
World. I am thread Hello World. I am thread Hello World. I
am thread Hello World. I am thread Hello World. I am
thread Hello World. I am thread Hello World. I am thread
02Hello World. I am thread Hello World. I am thread 13Hello
World. I am thread 5Hello World. I am thread Hello World. I
am thread 6Hello World. I am thread 47Hello World. I am
thread 8

910

Concurrency?

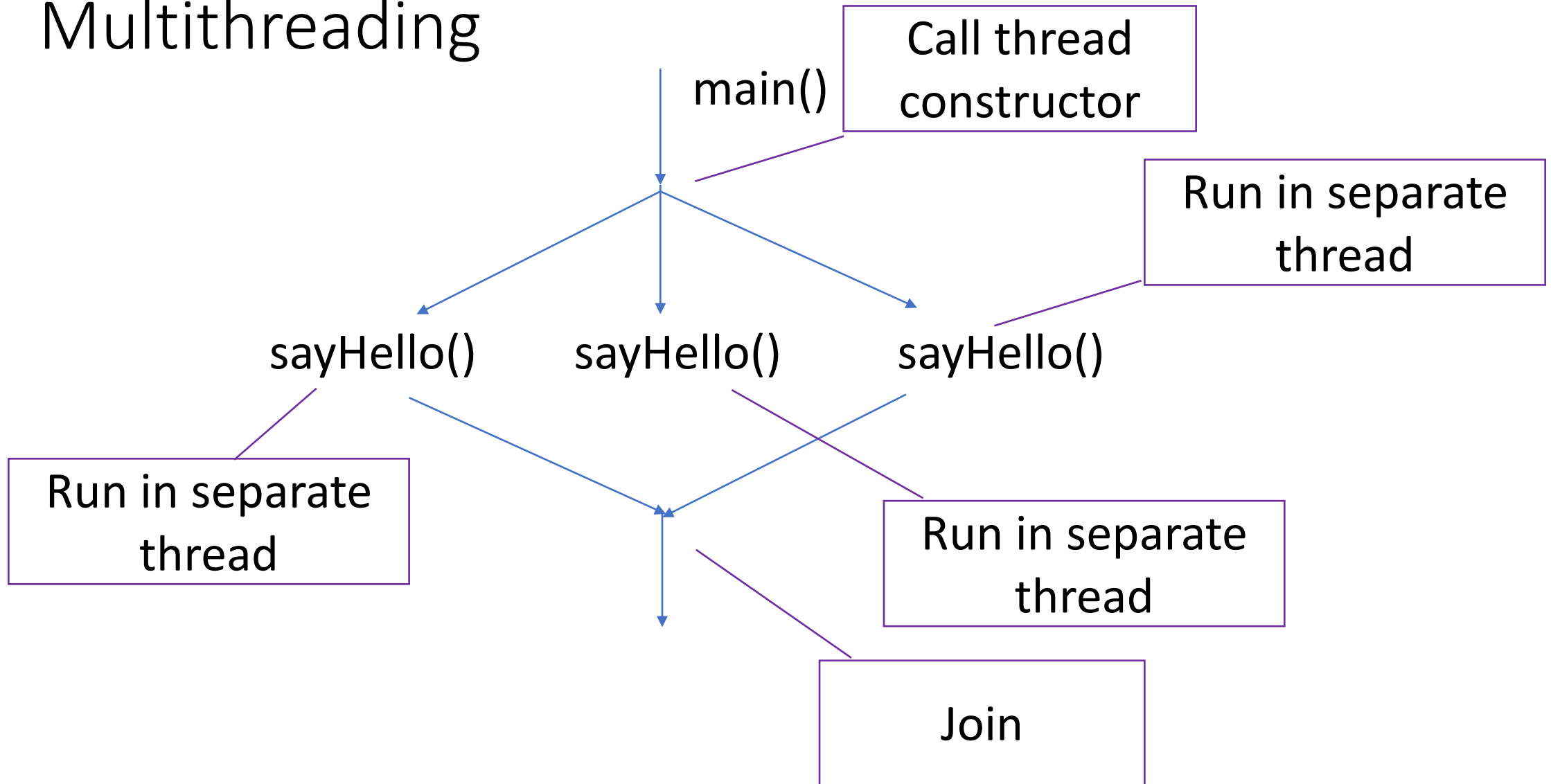
111213

14

Parallelism?

15

Multithreading



Why the Jumbled Output

```
void sayHello(int tnum) {  
    cout << "Hello World. I am thread " << tnum << endl;  
}  
  
int main() {  
    std::thread tid[16];  
  
    for (int i = 0; i < 16; ++i)  
        tid[i] = thread (sayHello, i);  
  
    for (int i = 0; i < 16; ++i)  
        tid[i].join();  
  
    return 0;  
}
```

Concurrency!

\$./a.out

Hello World. I am thread Hello World. I am thread Hello
World. I am thread Hello World. I am thread Hello World. I
am thread Hello World. I am thread Hello World. I am
thread Hello World. I am thread Hello World. I am thread
02Hello World. I am thread Hello World. I am thread 13Hello
World. I am thread 5Hello World. I am thread Hello World. I
am thread 6Hello World. I am thread 47Hello World. I am
thread 8

910

111213

14

15

Another Example

```
int value = 0;

int value = 0;

int main() {
    std::thread tid[16];

    for (int i = 0; i < 16; ++i)
        tid[i] = thread (sayHello, i);

    for (int i = 0; i < 16; ++i)
        tid[i].join();

    cout << "Final value is " << value << endl;

    return 0;
}
```


Example

./a.outHello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread Hello World. I am thread 5302Hello World. I am thread Hello World. I am thread 64Hello World. I am thread Hello World. I am thread 1Hello World. I am thread 789Value is Value is Value is Hello World. I am thread Value is 1011Value is Value is 1213Value is 14Value is Value is Value is 000150Value is Value is 00Value is Value is 0Value is 000Value is 000000

Final value is 1

Not Good!

Race condition

We will cover this next lecture

Review

- Process is an abstraction for resource allocation
- Thread is an abstraction for execution
- Concurrency vs Parallelism
- C++ threading library

Thank You!

Creative Commons BY-NC-SA 4.0 License



© Andrew Lumsdaine, 2017-2022

Except where otherwise noted, this work is licensed under

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

