CS 61C:
Great Ideas in Computer Architecture

Lecture 19:
Thread-Level Parallel Processing

Bernhard Boser & Randy Katz

http://inst.eecs.berkeley.edu/~cs61c

Agenda

- **MIMD - multiple programs simultaneously**
- Threads
- Parallel programming: OpenMP
- Synchronization primitives
- Synchronization in OpenMP
- And, in Conclusion …

Improving Performance

1. Increase clock rate f_s
   - Reached practical maximum for today's technology
   - < 5GHz for general purpose computers
2. Lower CPI (cycles per instruction)
   - SIMD, "instruction level parallelism"
3. Perform multiple tasks simultaneously
   - Multiple CPUs, each executing different program
   - Tasks may be related
     - E.g. each CPU performs part of a big matrix multiplication
   - or unrelated
     - E.g. distribute different web http requests over different computers
     - E.g. run ppt (view lecture slides) and browser (youtube) simultaneously
4. Do all of the above:
   - High f_s, SIMD, multiple parallel tasks

Today's lecture

New-School Machine Structures (It's a bit more complicated!)

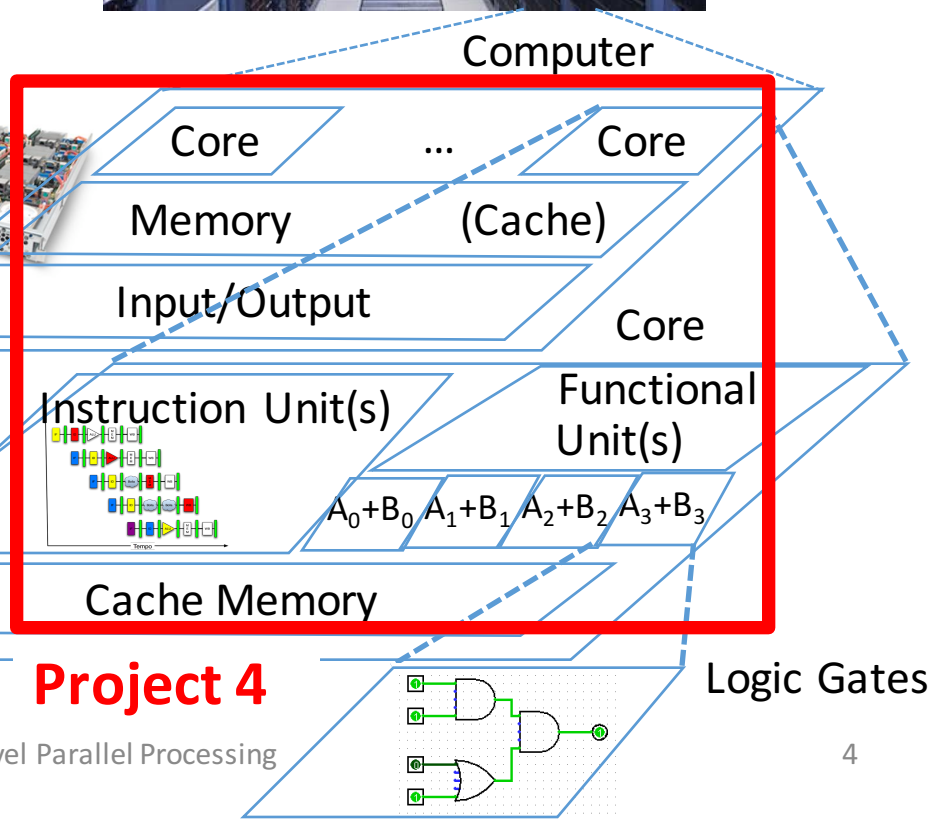
Software | *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates @ one time
- Programming Languages

Harness Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone

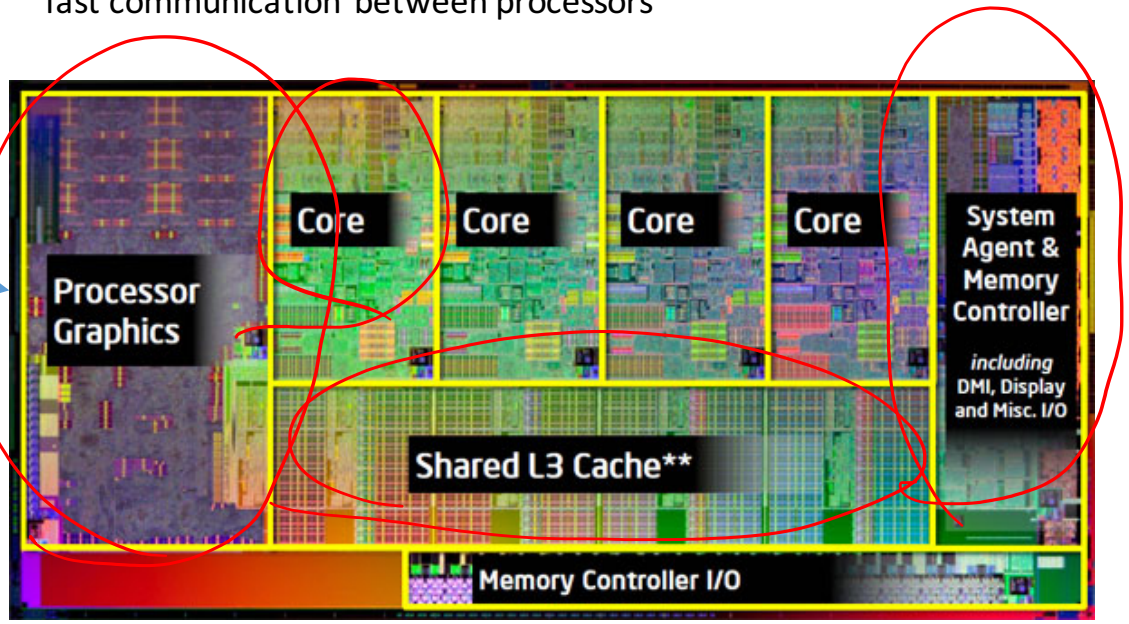


Parallel Computer Architectures

Several separate computers,
some means for communication
(e.g. Ethernet)

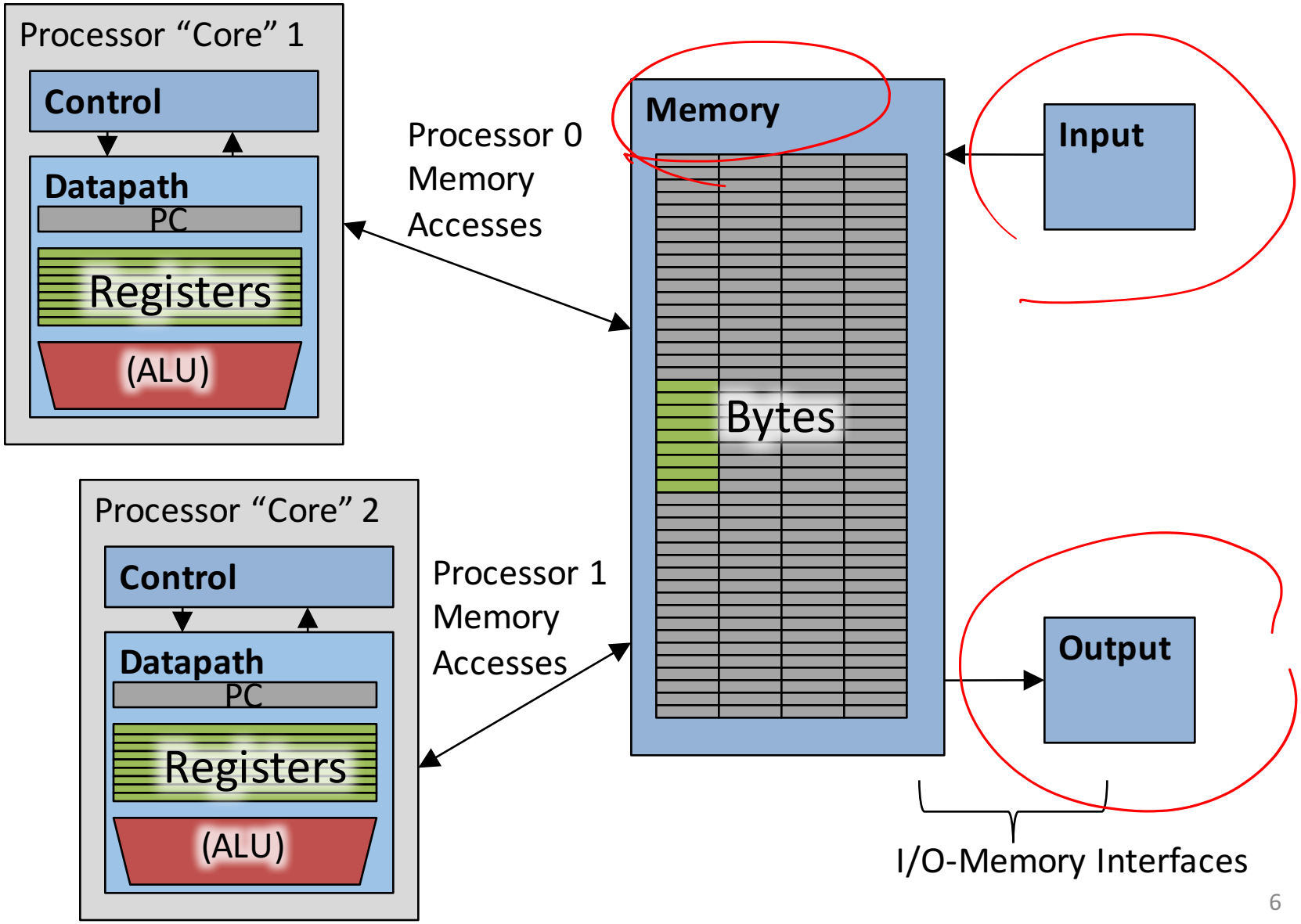
Massive array of computers,
fast communication between processors

GPU "graphics processing unit"



Multi-core CPU:
1 datapath in single chip
share L3 cache, memory, peripherals
Example: Hive machines

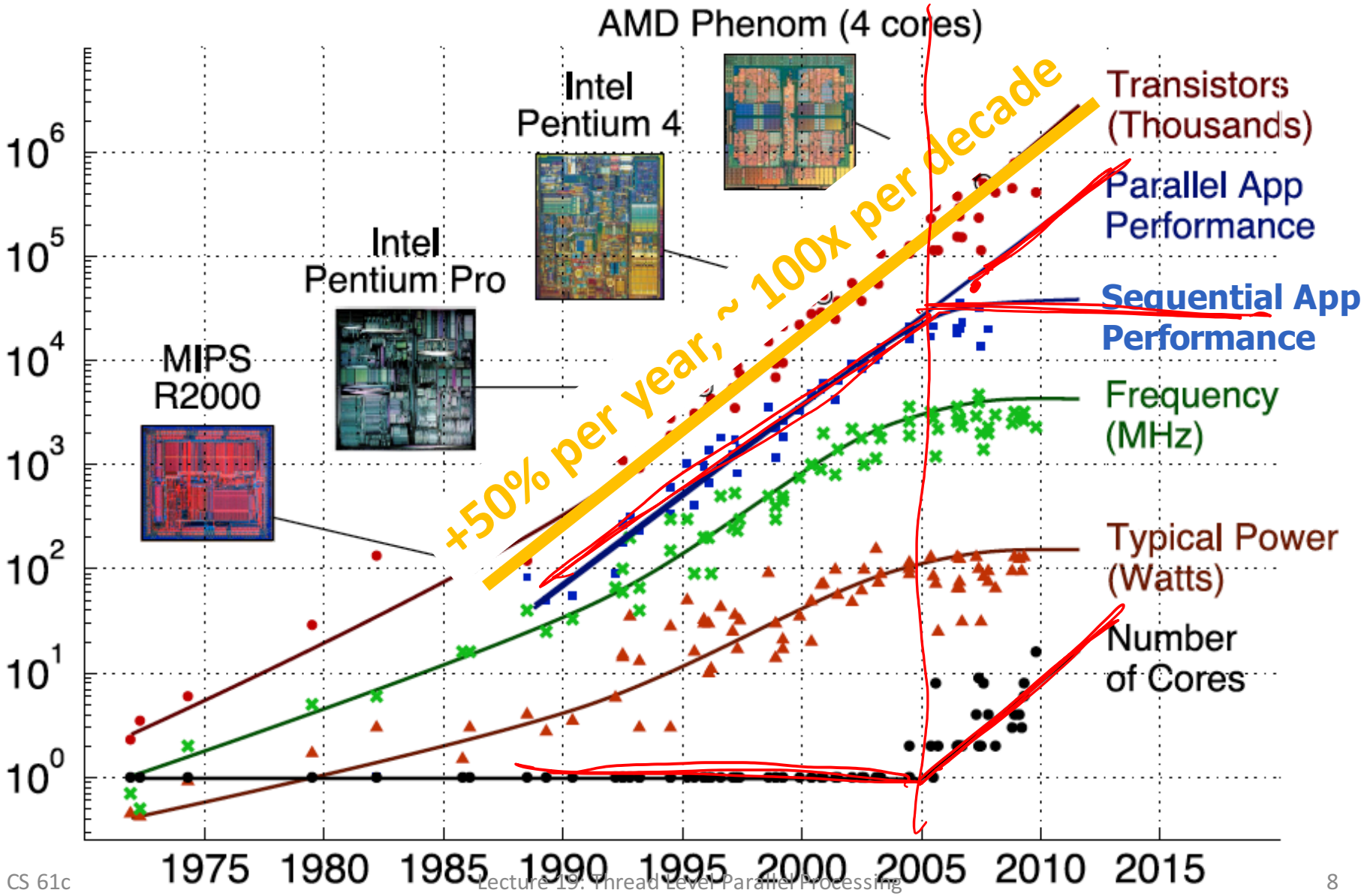
Example: CPU with 2 Cores



Multiprocessor Execution Model

- Each processor (core) executes its own instructions
- *Separate* resources (not shared)
  - Datapath (PC, registers, ALU)
  - Highest level caches (e.g. 1st and 2nd)
- ***Shared*** resources
  - Memory (DRAM)
  - Often 3rd level cache
    - Often on same silicon chip
    - But not a requirement
- Nomenclature
  - "Multiprocessor Microprocessor"
  - Multicore processor
    - E.g. 4 core CPU (central processing unit)
    - Executes 4 different instruction streams simultaneously

Transition to Multicore



Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond

Multiprocessor Execution Model

- Shared memory
  - Each "core" has access to the entire memory in the processor
  - Special hardware keeps caches consistent
  - Advantages:
    - Simplifies communication in program via shared variables
  - Drawbacks:
    - Does not scale well:
      - "Slow" memory shared by many "customers" (cores)
      - May become bottleneck (Amdahl's Law)
- Two ways to use a multiprocessor:
  - Job-level parallelism
    - Processors work on unrelated problems
    - No communication between programs
  - Partition work of single task between several cores
    - E.g. each performs part of large matrix multiplication

Parallel Processing

- It's difficult!
- It's inevitable
  - Only path to increase performance
  - Only path to lower energy consumption (improve battery life)
- In mobile systems (e.g. smart phones, tablets)
  - Multiple cores
  - Dedicated processors, e.g.
    - motion processor in iPhone
    - GPU (graphics processing unit)
- Warehouse-scale computers
  - multiple "nodes"
    - "boxes" with several CPUs, disks per box
  - MIMD (multi-core) and SIMD (e.g. AVX) in each node

Potential Parallel Performance
(assuming software can use it)

Year	Cores	SIMD bits /Core	Core * SIMD bits	Total, e.g. FLOPs/Cycle
2003	MIMD 2	SIMD 128	256	MIMD 4
2005	+2/ 4	2X/ 128	512	& SIMD 8
2007	2yrs 6	4yrs 128	768	12
2009	8	128	1024	16
2011	10	256	2560	40
2013	12	256	3072	48
2015	2.5X 14	8X 512	7168	20X 112
2017	16	512	8192	128
2019	18	1024	18432	288
2021	20	1024	20480	320

12 years

20 x in 12 years

$20^{1/12}$ = 1.28 x → 28% per year or 2x every 3 years!

IF (!) we can use it

Agenda

- MIMD - multiple programs simultaneously
- **Threads**
- Parallel programming: OpenMP
- Synchronization primitives
- Synchronization in OpenMP
- And, in Conclusion …

Programs Running on my Computer

```
  PID  TTY           TIME CMD
  220  ??         0:04.34 /usr/libexec/UserEventAgent  (Aqua)
  222  ??         0:10.60 /usr/sbin/distnoted  agent
  224  ??         0:09.11 /usr/sbin/cfprefsd  agent
  229  ??         0:04.71 /usr/sbin/usernoted
  230  ??         0:02.35 /usr/libexec/nsurlsessiond
  232  ??         0:28.68 /System/Library/PrivateFrameworks/CalendarAgent.framework/Executables/CalendarAgent
  234  ??         0:04.36 /System/Library/PrivateFrameworks/GameCenterFoundation.framework/Versions/A/gamed
  235  ??         0:01.90 /System/Library/CoreServices/cloudphotosd.app/Contents/MacOS/cloudphotosd
  236  ??         0:49.72 /usr/libexec/secinitd
  239  ??         0:01.66 /System/Library/PrivateFrameworks/TCC.framework/Resources/tccd
  240  ??         0:12.68 /System/Library/Frameworks/Accounts.framework/Versions/A/Support/accountsd
  241  ??         0:09.56 /usr/libexec/SafariCloudHistoryPushAgent
  242  ??         0:00.27 /System/Library/PrivateFrameworks/CallHistory.framework/Support/CallHistorySyncHelper
  243  ??         0:00.74 /System/Library/CoreServices/mapspushd
  244  ??         0:00.79 /usr/libexec/fmfd
  246  ??         0:00.09 /System/Library/PrivateFrameworks/AskPermission.framework/Versions/A/Resources/askpermissiond
  248  ??         0:01.03 /System/Library/PrivateFrameworks/CloudDocsDaemon.framework/Versions/A/Support/bird
  249  ??         0:02.50 /System/Library/PrivateFrameworks/IDS.framework/identityservicesd.app/Contents/MacOS/identityservicesd
  250  ??         0:04.81 /usr/libexec/secd
  254  ??         0:24.01 /System/Library/PrivateFrameworks/CloudKitDaemon.framework/Support/cloudd
  258  ??         0:04.73 /System/Library/PrivateFrameworks/TelephonyUtilities.framework/callservicesd
  267  ??         0:02.15 /System/Library/CoreServices/AirPlayUIAgent.app/Contents/MacOS/AirPlayUIAgent  --launchd
  271  ??         0:03.91 /usr/libexec/nsurlstoraged
  274  ??         0:00.90 /System/Library/PrivateFrameworks/CommerceKit.framework/Versions/A/Resources/storeaccountd
  282  ??         0:00.09 /usr/sbin/pboard
  283  ??         0:00.90
/System/Library/PrivateFrameworks/InternetAccounts.framework/Versions/A/XPCServices/com.apple.internetaccounts.xpc/Contents/MacOS/com.apple.internetaccounts
  285  ??         0:04.72 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/ATS.framework/Support/fontd
  291  ??         0:00.25 /System/Library/Frameworks/Security.framework/Versions/A/Resources/CloudKeychainProxy.bundle/Contents/MacOS/CloudKeychainProxy
  292  ??         0:09.54 /System/Library/CoreServices/CoreServicesUIAgent.app/Contents/MacOS/CoreServicesUIAgent
  293  ??         0:00.29
/System/Library/PrivateFrameworks/CloudPhotoServices.framework/Versions/A/Frameworks/CloudPhotoServicesConfiguration.framework/Versions/A/XPCServices/com.ap
ple.CloudPhotosConfiguration.xpc/Contents/MacOS/com.apple.CloudPhotosConfiguration
  297  ??         0:00.84 /System/Library/PrivateFrameworks/CloudServices.framework/Resources/com.apple.sbd
  302  ??         0:26.11 /System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
  303  ??         0:09.55 /System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer
```

… 156 total at this moment

How does my laptop do this?

Imagine doing 156 assignments all at the same time!

Threads

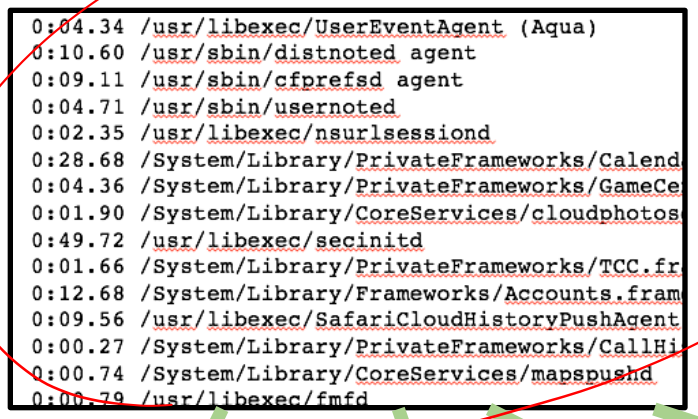
- Sequential flow of instructions that performs some task
  - Up to now we just called this a "program"
- Each thread has a
  - Dedicated PC (program counter)
  - Separate registers
  - Accesses the shared memory
- Each processor provides one (or more)
  - *hardware* threads (or *harts*) that actively execute instructions
  - Each core executes one "*hardware* thread"
- Operating system multiplexes multiple
  - *software* threads onto the available hardware threads
  - all threads except those mapped to hardware threads are waiting

Operating System Threads

Give illusion of many "simultaneously" active threads

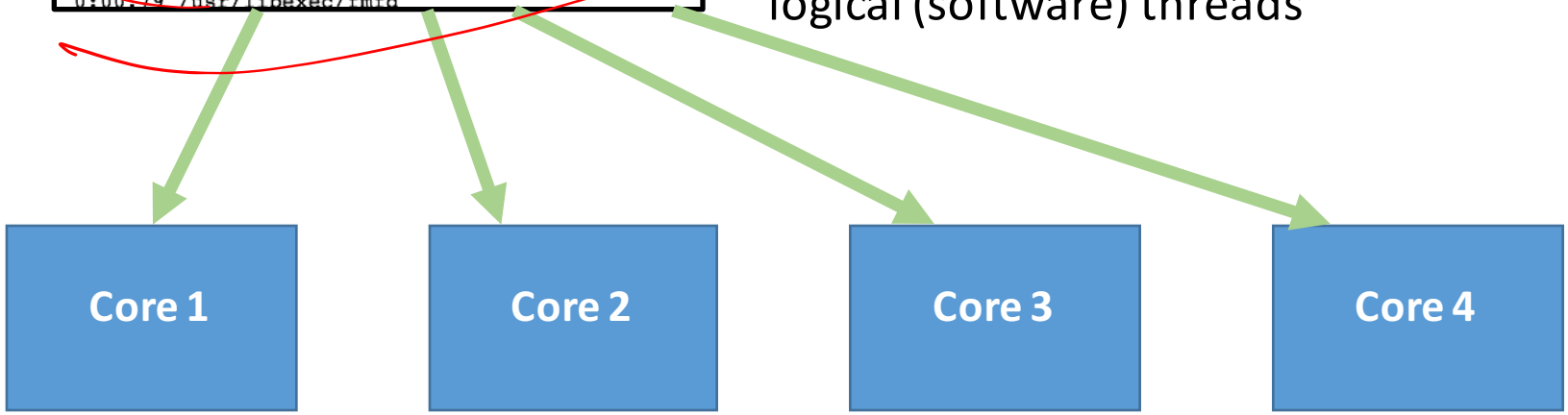
1. Multiplex software threads onto hardware threads:
   a) Switch out blocked threads (e.g. cache miss, user input, network access)
   b) Timer (e.g. switch active thread every 1ms)
2. Remove a software thread from a hardware thread by
   i. interrupting its execution
   ii. saving its registers and PC to memory
3. Start executing a different software thread by
   i. loading its previously saved registers into a hardware thread's registers
   ii. jumping to its saved PC

Example: 4 Cores



Thread pool:
List of threads competing for processor

OS maps threads to cores and schedules logical (software) threads

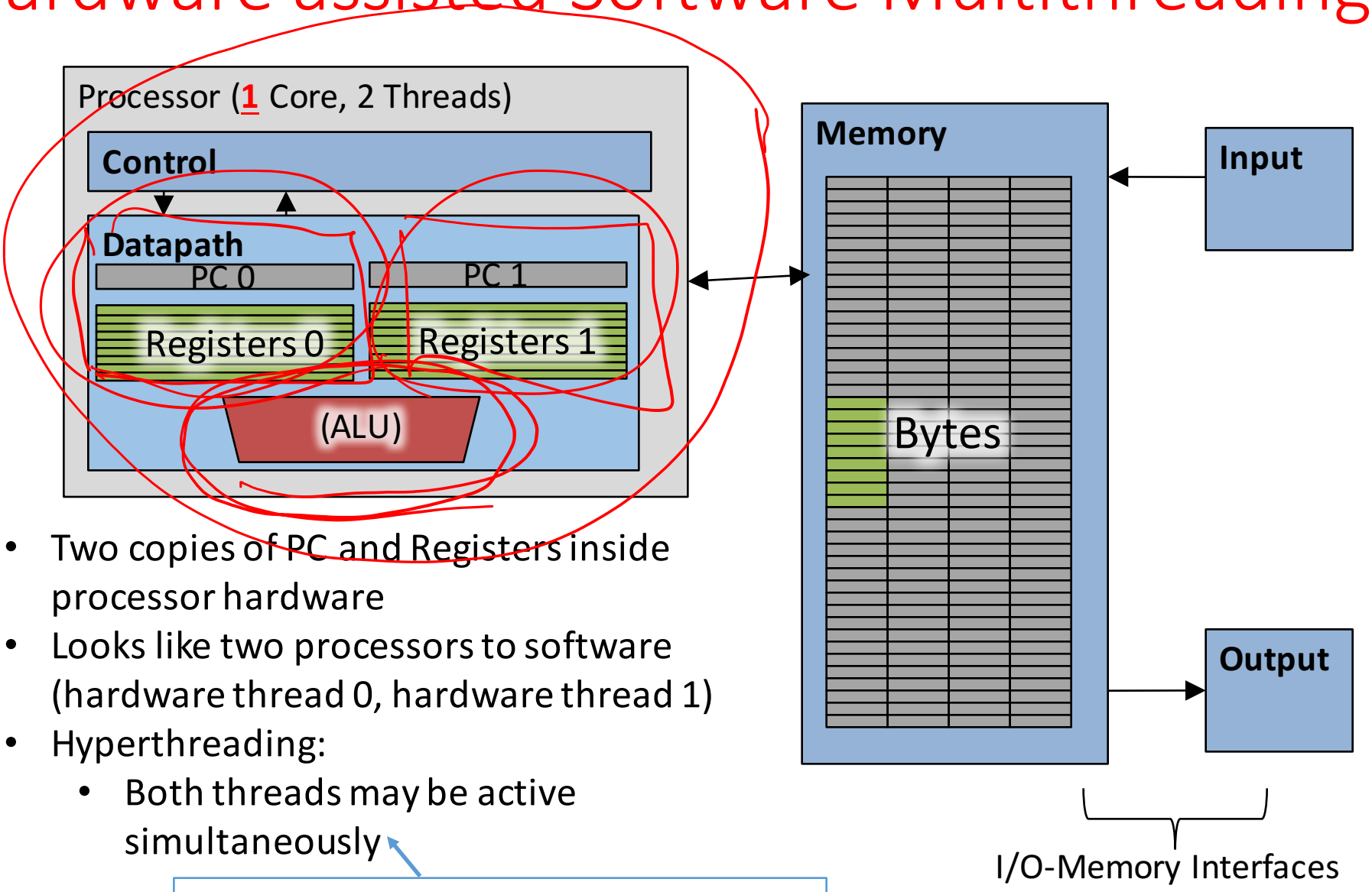


Each "Core" actively runs 1 program at a time

Multithreading

- Typical scenario:
  - Active thread encounters cache miss
  - Active thread waits $\sim$ 1000 cycles for data from DRAM
  - → switch out and run different thread until data available
- Problem
  - Must save current thread state and load new thread state
    - PC, all registers (could be many, e.g. AVX)
  - → must perform switch in $\ll$ 1000 cycles
- Can hardware help?
  - Moore's law: transistors are plenty

Hardware assisted Software Multithreading



- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Hyperthreading:
  - Both threads may be active simultaneously

Note: presented incorrectly in the lecture

Multithreading

- Logical threads
  - ≈ 1% more hardware, ≈ 10% (?) better performance
    - Separate registers
    - Share datapath, ALU(s), caches
- Multicore
  - => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
- Modern machines do both
  - Multiple cores with multiple threads per core

Bernhard's Laptop

```
$ sysctl -a | grep hw

hw.physicalcpu:                    2
hw.logicalcpu:                     4
hw.l1icachesize:              32,768
hw.l1dcachesize:              32,768
hw.l2cachesize:              262,144
hw.l3cachesize:            3,145,728
```

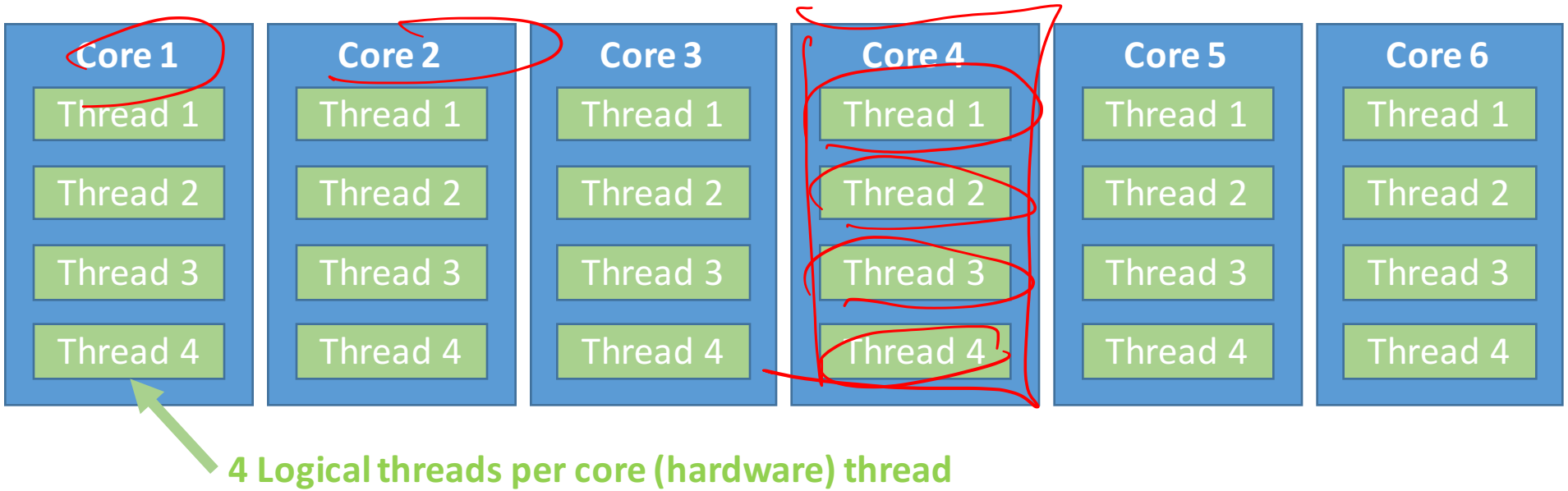
- 2 Cores
- 4 Threads total

Example: 6 Cores, 24 Logical Threads

```
0:04.34 /usr/libexec/UserEventAgent (Aqua)
0:10.60 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/cfprefsd agent
0:04.71 /usr/sbin/usernoted
0:02.35 /usr/libexec/nsurlsessiond
0:28.68 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameCe
0:01.90 /System/Library/CoreServices/cloudphotos
0:49.72 /usr/libexec/secinitd
0:01.66 /System/Library/PrivateFrameworks/TCC.fr
0:12.68 /System/Library/Frameworks/Accounts.fram
0:09.56 /usr/libexec/SafariCloudHistoryPushAgent
0:00.27 /System/Library/PrivateFrameworks/CallHi
0:00.74 /System/Library/CoreServices/mapspushd
0:00.79 /usr/libexec/fmfd
```

Thread pool:
List of threads competing for processor

OS maps threads to cores and schedules logical (software) threads



Agenda

- MIMD - multiple programs simultaneously
- Threads
- **Parallel programming: OpenMP**
- Synchronization primitives
- Synchronization in OpenMP
- And, in Conclusion …

Languages supporting Parallel Programming

ActorScript	Concurrent Pascal	JoCaml	Orc
Ada	Concurrent ML	Join	Oz
Afnix	Concurrent Haskell	Java	Pict
Alef	Curry	Joule	Reia
Alice	CUDA	Joyce	SALSA
APL	E	LabVIEW	Scala
Axum	Eiffel	Limbo	SISAL
Chapel	Erlang	Linda	SR
Cilk	Fortan 90	MultiLisp	Stackless Python
Clean	Go	Modula-3	SuperPascal
Clojure	Io	Occam	VHDL
Concurrent C	Janus	occam-π	XC

Which one to pick?

Why so many parallel programming languages?

- Piazza question:
  - Why “intrinsics”?
  - TO Intel: fix your #()&$! Compiler!
- It’s happening ... but
  - SIMD features are continually added to compilers (Intel, gcc)
  - Intense area of research
  - Research progress:
    - 20+ years to translate C into good (fast!) assembly
    - How long to translate C into good (fast!) parallel code?
      - General problem is very hard to solve
      - Present state: specialized solutions for specific cases
      - **Your opportunity to become famous!**

Parallel Programming Languages

- Number of choices is indication of
  - No universal solution
    - Needs are very problem specific
  - E.g.
    - Scientific computing (matrix multiply)
    - Webserver: handle many unrelated requests simultaneously
    - Input / output: it's all happening simultaneously!
- Specialized languages for different tasks
  - Some are easier to use (for some problems)
  - None is particularly "easy" to use
- 61C
  - Parallel language examples for high-performance computing
  - OpenMP

Parallel Loops

- Serial execution:

```
for (int i=0; i<100; i++) {
        …
}
```

- Parallel Execution:

`for (int i=0; i<25; i++) { … }`	`for (int i=25; i<50; i++) { … }`	`for (int i=50; i<75; i++) { … }`	`for (int i=75; i<100; i++) { … }`

Parallel **for** in OpenMP

```
#include <omp.h>

#pragma omp parallel for
for (int i=0; i<100; i++) {
        …
}
```

OpenMP Example

```
#include <stdio.h>
#include <omp.h>

// gcc-5 -fopenmp for.c

int main(void) {
    omp_set_num_threads(4);
    int a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int N = sizeof(a)/sizeof(int);

    #pragma omp parallel for
        for (int i=0;  i<N;  i++) {
            printf("thread %d, i = %2d\n",
                    omp_get_thread_num(), i);
            a[i] = a[i] + 100*omp_get_thread_num();
        }

    for (int i=0;  i<N;  i++) printf("%d ", a[i]);
    printf("\n");
}
```

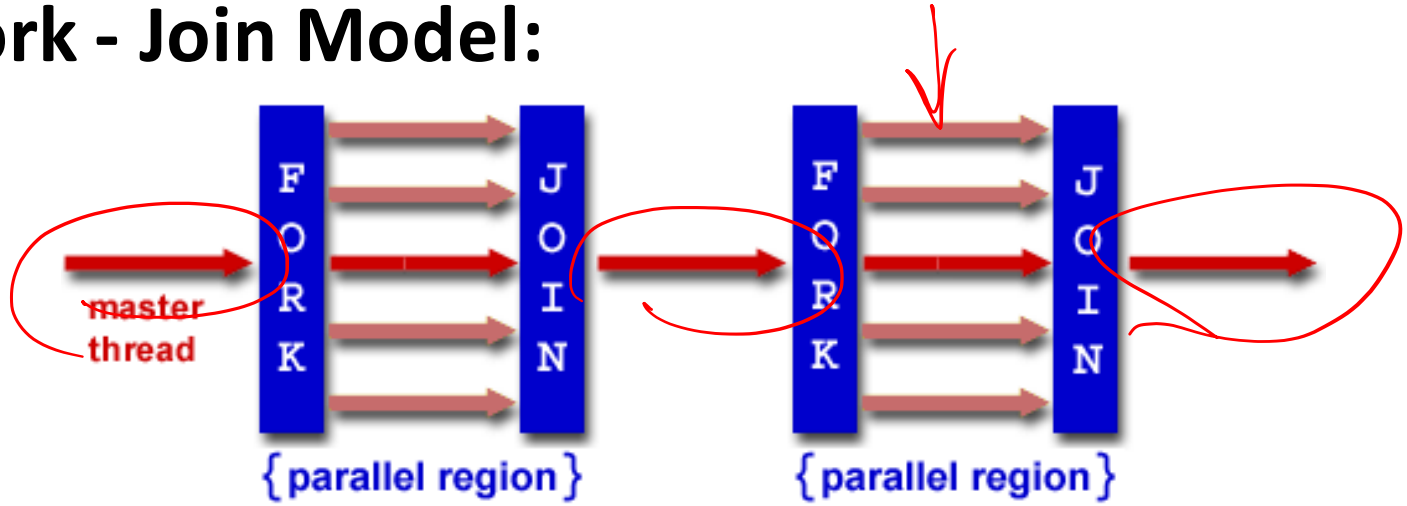
```
$ gcc-5 -fopenmp for.c;./a.out
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
01 02 03 14 15 16 27 28 39 40
```

OpenMP

- C extension: no new language to learn
- Multi-threaded, shared-memory parallelism
  - Compiler Directives, **`#pragma`**
  - Runtime Library Routines, **`#include <omp.h>`**
- **`#pragma`**
  - Ignored by compilers unaware of OpenMP
  - Same source for multiple architectures
    - E.g. same program for 1 & 16 cores
- Only works with shared memory

OpenMP Programming Model

- **Fork - Join Model:**



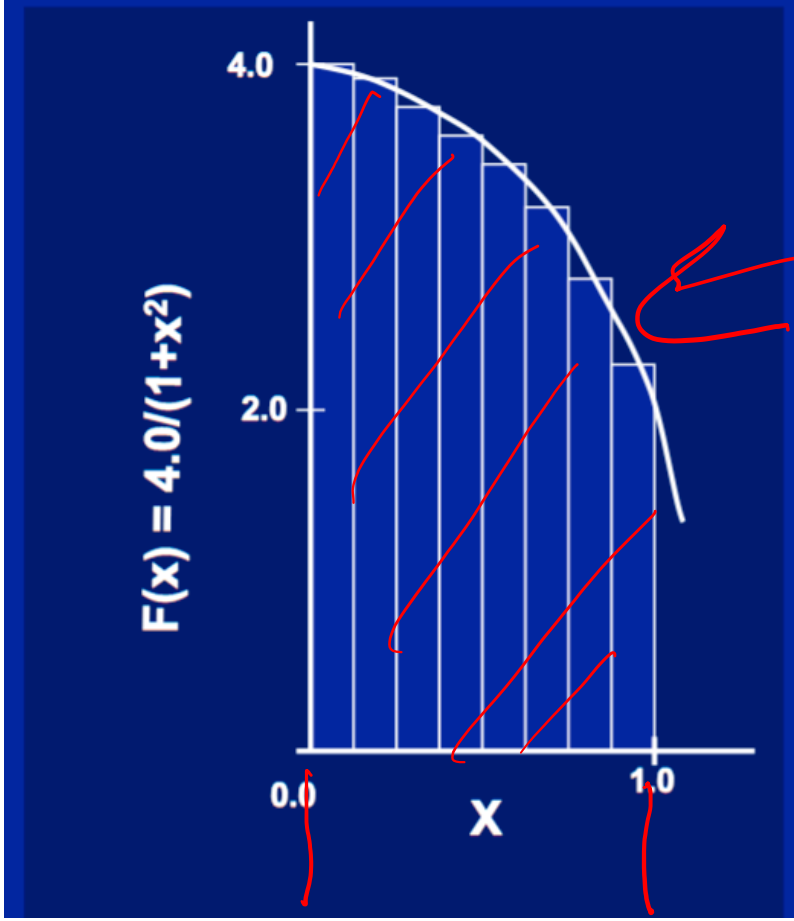
- OpenMP programs begin as single process (*master thread*)
  - Sequential execution
- When parallel region is encountered
  - Master thread "forks" into team of parallel threads
  - Executed simultaneously
  - At end of parallel region, parallel threads "join", leaving only master thread
- Process repeats for each parallel region
  - Amdahl's law?

What Kind of Threads?

- OpenMP threads are operating system (software) threads.
- OS will multiplex requested OpenMP threads onto available hardware threads.
- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing.
- But other tasks on machine can also use hardware threads!
- Be “careful” (?) when timing results for project 4!
  - 5AM?
  - Job queue?

Example 2: computing π

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

Sequential π

```
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

```
pi = 3.142425985001
```

- Resembles π, but not very accurate
- Let's increase **`num_steps`** and parallelize

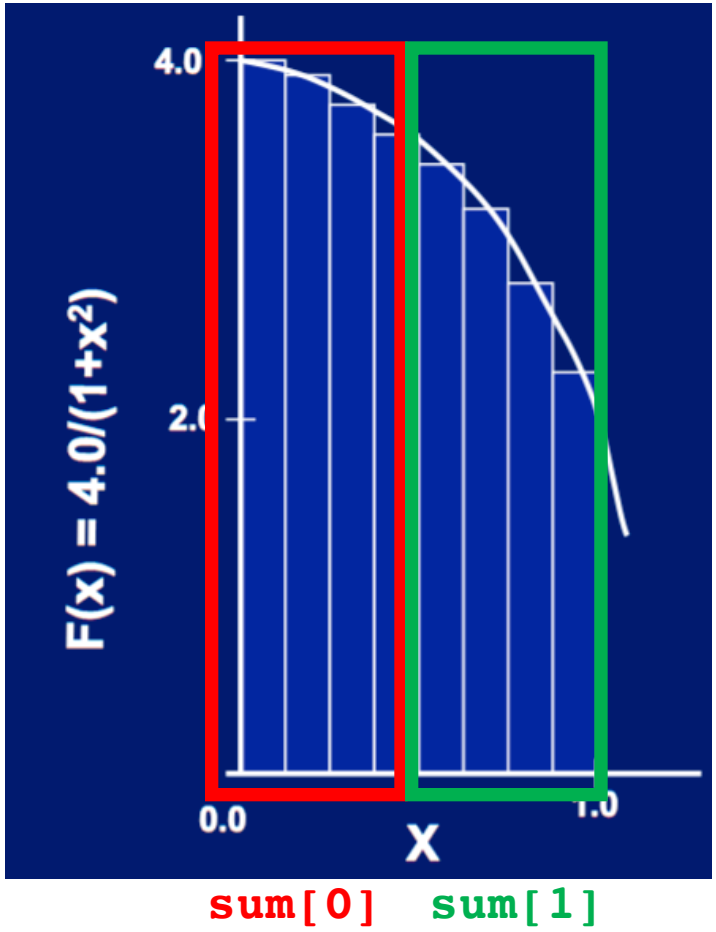
Parallelize (1) …

```
#include <stdio.h>

void main () {
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum = 0.0;
#pragma parallel for
    for (int i=0; i<num_steps; i++) {
        double x = (i+0.5) *step;
        sum += 4.0*step/(1.0+x*x);
    }
    printf ("pi = %6.12f\n", sum);
}
```

- Problem: each threads needs access to the shared variable **`sum`**
- Code runs sequentially …

Parallelize (2) …



1. Compute
   `sum[0]` and **`sum[2]`**
   in parallel

2. Compute
   `sum = sum[0] + sum[1]`
   sequentially

Parallel π

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

Trial Run

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 10;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

```
i =   1,   id =   1
i =   0,   id =   0
i =   2,   id =   2
i =   3,   id =   3
i =   5,   id =   1
i =   4,   id =   0
i =   6,   id =   2
i =   7,   id =   3
i =   9,   id =   1
i =   8,   id =   0
pi = 3.142425985001
```

Scale up: **num_steps = 10^6**

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 1000000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
            // printf("i =%3d,  id =%3d\n", i, id);
        }
    }
    double pi = 0;
    for (int i=0; i<NUM_THREADS; i++) pi += sum[i];
    printf ("pi = %6.12f\n", pi);
}
```

pi = 3.141592653590

You verify how many digits are correct …

Can we Parallelize Computing **sum**

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

Always looking for ways to beat Amdahl's Law …

Summation inside parallel section

- Insignificant speedup in this example, but …
- **`pi = 3.138450662641`**
- Wrong! And value changes between runs?!
- What's going on?

Your Turn

What are the possible values of `*($s0)` after executing this code by 2 ***concurrent*** threads?

```
#  *($s0) = 100
lw   $t0,0($s0)
addi $t0,$t0,1
sw   $t0,0($s0)
```

Answer	*($s0)
A	100 or 101
B	101
C	101 or 102
D	100 or 101 or 102
E	100 or 101 or 102 or 103

Your Turn

What are the possible values of **`*($s0)`** after executing this code by 2 ***concurrent*** threads?

```
# *($s0) = 100
lw   $t0,0($s0)
addi $t0,$t0,1
sw   $t0,0($s0)
```

Answer	*($s0)
C	101 or 102

- 102 if the threads enter code section sequentially
- 101 if both execute **`lw`** before either runs **`sw`**
- one thread sees “stale” data

What's going on?

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

- Operation is really

  `pi = pi + sum[id]`
- What if >1 threads reads current (same) value of **`pi`**, computes the sum, and stores the result back to **`pi`**?
- Each processor reads same intermediate value of **`pi`**!
- Result depends on who gets there when
  - A "race" → result is **not deterministic**

Agenda

- MIMD - multiple programs simultaneously
- Threads
- Parallel programming: OpenMP
- **Synchronization primitives**
- Synchronization in OpenMP
- And, in Conclusion …

Synchronization

- Problem:
  - Limit access to shared resource to 1 actor at a time
  - E.g. only 1 person permitted to edit a file at a time
    - otherwise changes by several people get all mixed up

- Solution:

- Take turns:
  - Only one person get's the microphone & talks at a time
  - Also good practice for classrooms, btw …

Locks

- Computers use locks to control access to shared resources
  - Serves purpose of microphone in example
  - Also referred to as “semaphore”

- Usually implemented with a variable
  - **`int lock;`**
    - 0 for unlocked
    - 1 for locked

Synchronization with locks

```
// wait for lock released
while (lock != 0) ;
// lock == 0 now (unlocked)

// set lock
lock = 1;

    // access shared resource ...
    // e.g. pi
    // sequential execution! (Amdahl ...)

// release lock
lock = 0;
```

Lock Synchronization

Thread 1

```
while (lock !=
```

```
= 0) ;
```

set,

```
lock = 1;
```

ney got

and set the lock!

```
// critical section
```

```
lock
```

Try as you want, this problem has no solution, not even at the assembly level.
Unless we introduce new instructions, that is!

Hardware Synchronization

- Solution:
  - Atomic read/write
  - Read & write in single instruction
    - No other access permitted between read and write
  - Note:
    - Must use *shared memory* (multiprocessing)
- Common implementations:
  - Atomic swap of register ↔ memory
  - Pair of instructions for "linked" read and write
    - write fails if memory location has been "tampered" with after linked read
    - MIPS uses this solution

MIPS Synchronization Instructions

- *Load linked:* **`ll $rt, off($rs)`**

  - Reads memory location (like **`lw`**)
  - Also sets (hidden) "link bit"
  - Link bit is reset if memory location (**`off($rs)`**) is accessed

- *Store conditional:* **`sc $rt, off($rs)`**

  - Stores **`off($rs) = $rt`** (like **`sw`**)
  - Sets **`$rt=1`** (success) if link bit is set
    - i.e. no (other) process accessed **`off($rs)`** since **`ll`**
  - Sets **`$rt=0`** (failure) otherwise
  - Note: **`sc`** *clobbers* **`$rt`**, i.e. changes its value

Lock Synchronization

$t0 = 1 before calling **ll**: minimize time between **ll** and **sc**

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location $s1)

```
Try: addiu $t0,$zero,1
     ll   $t1,0($s1)
     bne  $t1,$zero,Try
     sc   $t0,0($s1)
     beq  $t0,$zero,Try
Locked:
     # critical section

Unlock:
     sw $zero,0($s1)
```

Try again if **sc** failed (another thread executed **sc** since above **ll**)

Agenda

- MIMD - multiple programs simultaneously
- Threads
- Parallel programming: OpenMP
- Synchronization primitives
- **Synchronization in OpenMP**
- And, in Conclusion …

OpenMP Locks

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(void) {
    omp_lock_t lock;
    omp_init_lock(&lock);

#pragma omp parallel
    {
        int id = omp_get_thread_num();

        // parallel section
        // ...

        omp_set_lock(&lock);
        // start sequential section
        // ...
        printf("id = %d\n", id);

        // end sequential section
        omp_unset_lock(&lock);

        // parallel section
        // ...

    }
    omp_destroy_lock(&lock);
}
```






Synchronization in OpenMP

- Typically are used in libraries of higher level parallel programming constructs
- E.g. OpenMP offers **`$pragma`**s for common cases:
  - critical
  - atomic
  - barrier
  - ordered
- OpenMP offers many more features
  - see online documentation
  - or tutorial at
    - http://openmp.org/mp-documents/omp-hands-on-SC08.pdf

OpenMP critical

```
#include <stdio.h>
#include <omp.h>

void main () {
    const int NUM_THREADS = 1000;
    const long num_steps = 100000;
    double step = 1.0/((double)num_steps);
    double sum[NUM_THREADS];
    for (int i=0; i<NUM_THREADS; i++) sum[i] = 0;
    double pi = 0;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
    {
        int id = omp_get_thread_num();
        for (int i=id; i<num_steps; i+=NUM_THREADS) {
            double x = (i+0.5) *step;
            sum[id] += 4.0*step/(1.0+x*x);
        }
#pragma omp critical
        pi += sum[id];
    }
    printf ("pi = %6.12f\n", pi);
}
```

The Trouble with Locks …

- … is ***dead-locks***
- Consider 2 cooks sharing a kitchen
  - Each cooks a meal that requires salt and pepper (locks)
  - Cook 1 grabs salt
  - Cook 2 grabs pepper
  - Cook 1 notices s/he needs pepper
    - it's not there, so s/he waits
  - Cook 2 realizes s/he needs salt
    - it's not there, so s/he waits
- A not so common cause of cook starvation
  - But deadlocks are possible in parallel programs
  - Very difficult to debug
    - **`malloc/free`** is easy …

Agenda

- MIMD - multiple programs simultaneously
- Threads
- Parallel programming: OpenMP
- Synchronization primitives
- Synchronization in OpenMP
- **And, in Conclusion …**

And in Conclusion, …

- Sequential software execution speed is limited
- Parallel processing is the only path to higher performance
  - SIMD: instruction level parallelism
    - Implemented in all high performance CPUs today (x86, ARM, …)
    - Partially supported by compilers
  - MIMD: thread level parallelism
    - Multicore processors
    - Supported by Operating Systems (OS)
    - Requires programmer intervention to exploit at single program level
      - E.g. OpenMP
  - SIMD & MIMD for maximum performance
- Synchronization
  - Requires hardware support: specialized assembly instructions
  - Typically use higher-level support
  - Beware of deadlocks