CS 61C: Great Ideas in Computer Architecture

Lecture 19: *Thread-Level Parallel Processing*

Krste Asanović & Randy H. Katz

http://inst.eecs.berkeley.edu/~cs61c/fa17

Agenda

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

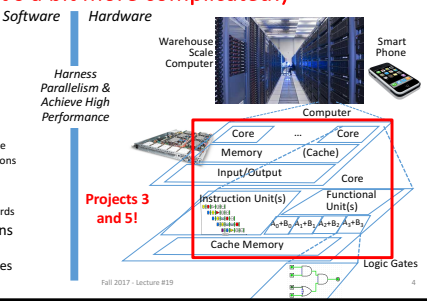
Improving Performance

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 pptx (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!)

- 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



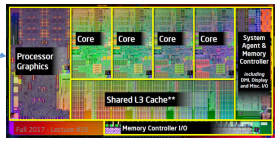
Parallel Computer Architectures

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

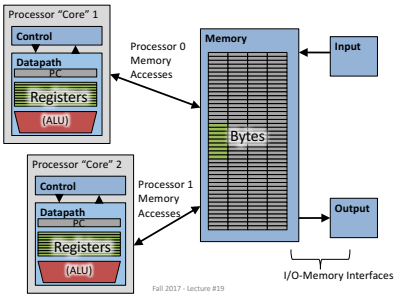
Massive array of computers, fast communication between processors

GPU "graphics processing unit"

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



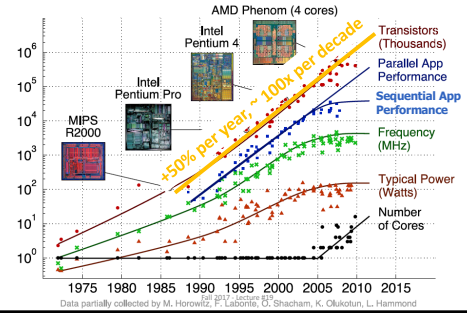
Example: CPU with Two Cores



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., four core CPU (central processing unit)
    - Executes four different instruction streams simultaneously

Transition to Multicore



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

	Pixel 2	iPhone 8
Size	5.74 x 2.74 x 0.31 inches	5.45 x 2.65 x 0.29 inches
Weight	143 grams (5.04 ounces)	148 grams (5.22 ounces)
Screen	5-inch AMOLED display	4.7-inch Retina HD LCD-backlit widescreen
Resolution	1,920 x 1,080 pixels (441 ppi)	1,334 x 750 pixels (326 pixels-per-inch)
OS	Android 8.0	iOS 11
Storage	64GB, 128GB	64GB, 256GB
MicroSD card slot	No	No
NFC support	Yes	Yes, Apple Pay only
Processor	Snapdragon 835, with Adreno 540	A11 Bionic with 64-bit architecture, M11 motion coprocessor
RAM	4GB	2GB
Connectivity	GSM, CDMA, HSPA, EVDO, LTE, 802.11a/b/g/n/ac Wi-Fi	4G LTE, GSM, CDMA, HSPA+, 802.11a/b/g/n/ac Wi-Fi
Camera	12.2 MP rear, 8 MP HD front	12.2 MP rear, 8 MP HD front
Video	Up to 4K at 30fps, 1080p at 120fps, 720p at 240fps	Up to 4K at 60fps, 1080p at 240fps

Pixel 2 vs. iPhone 8

Qualcomm Technologies leading with 10nm

Kryo 280 [edit]

A new generation of this microarchitecture, named Kryo 280, was announced along with the Snapdragon 835 chipset in November 2016.[3] The Kryo 280 CPU core is not a derivative of the original Kryo, but rather is a customized derivative of the ARM's Cortex-A73.[4] The new core improves integer instructions per clock (+17%), while having much lower (-32%) performance at floating point math relative to the original Kryo.[4]

Overview [edit]

- 32 KiB + 32 KiB L1 cache
- 2 MiB L2 cache (performance cluster only)
- Core performance: ~6.35 DMIPS/MHz

2.35Ghz + 1.9Ghz, 64Bit Octa-Core

Adreno 540		ALUs 256	nm 10	MHz 710	?	?	?	GFlops 567[51]					Snapdragon 835 (MSM8998)

Pixel 2 vs. iPhone 8

Design [edit]

The A11 features an Apple-designed 64-bit ARMv8-A six-core CPU, with two high-performance cores, called **Monsoon**, and four energy-efficient cores, called **Mistral**.[1][5][3] The A11 uses a new second-generation performance controller, which permits the A11 to use all six cores simultaneously,[7] unlike its predecessor the A10. The A11 also integrates an Apple-designed three-core graphics processing unit (GPU) with 30% faster graphics performance than the A10.[5] Embedded in the A11 is the M11 motion coprocessor.[8] The A11 includes a new image processor which supports computational photography functions such as lighting estimation, wide color capture, and advanced pixel processing.[5] The A11 also includes dedicated neural network hardware that Apple calls a "Neural Engine", which can perform up to 600 billion operations per second, used for Face ID, Animoji and other machine learning tasks.[7]

The A11 is manufactured by TSMC using a 10 nm FinFET process[1] and contains 4.3 billion transistors[6] on a die 87.66 mm^2 in size, 41% smaller than the A10.[9] It is manufactured in a package on package (PoP) together with 2 GB of LPDDR4X memory in the iPhone 8[2] and 3 GB of LPDDR4X memory in the iPhone 8 Plus and iPhone X.[9][10]

Geekbench Browser — Geekbench 4 — Geekbench 3

Single Core | Multi-Core | RenderScript

Device	Score
Samsung Galaxy Note 8 Samsung Exynos 8895 Octa @ 1.7 GHz	6479
Samsung Galaxy S8 Samsung Exynos 8895 Octa @ 1.7 GHz	6437
Samsung Galaxy S8+ Samsung Exynos 8895 Octa @ 1.7 GHz	6406
Samsung Galaxy Note 8 Qualcomm MSM8998 Snapdragon 835 @ 1.9 GHz	6306
Xiaomi Mi 6 Qualcomm MSM8998 Snapdragon 835 @ 1.9 GHz	6053
Samsung Galaxy S8+ Qualcomm MSM8998 Snapdragon 835 @ 1.9 GHz	6045
Huawei Honor V9 Hisilicon Kirin 960 @ 1.8 GHz	6024
Huawei P10 Hisilicon Kirin 960 @ 1.8 GHz	6000
Samsung Galaxy S8 Qualcomm MSM8998 Snapdragon 835 @ 1.9 GHz	5983

vs.

Geekbench Browser — Geekbench 4 — Geekbench 3

Single Core | Multi-Core | Metal

Device	Score
iPhone 8 Plus Apple A11 Bionic @ 2.4 GHz	10122
iPhone 8 Apple A11 Bionic @ 2.4 GHz	10030
iPhone X Apple A11 Bionic @ 2.4 GHz	9939
iPad Pro (10.5-inch) Apple A10X Fusion @ 2.3 GHz	9249
iPad Pro (12.9-inch 2nd Generation) Apple A10X Fusion @ 2.3 GHz	9236
iPhone 7 Apple A10 Fusion @ 2.3 GHz	5753
iPhone 7 Plus Apple A10 Fusion @ 2.3 GHz	5747
iPad Pro (12.9-inch) Apple A9X @ 2.3 GHz	5017
iPad Pro (9.7-inch) Apple A9X @ 2.3 GHz	4905

Multiprocessor Execution Model

- Shared memory
  - Each "core" has access to the entire memory in the processor
  - Special hardware keeps caches consistent (next lecture!)
  - 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, image processor, neural processor in iPhone 8 + X
    - GPU (graphics processing unit)
- Warehouse-scale computers (next week!)
  - 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 & SIMD 4
2005	+2/ 4	2X/ 128	512	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/UNetNextAgent (Aqua)
  222 ??   0:16.60 /usr/sbin/distnotad agent
  224 ??   0:09.11 /usr/sbin/cfprefsd agent
  225 ??   0:00.41 /usr/libexec/UserEvent
  228 ??   0:02.25 /usr/libexec/nsurlsessiond
  232 ??   0:28.68 /System/Library/PrivateFrameworks/CalendarAgent.framework/Executables/CalendarAgent
  234 ??   0:04.36 /System/Library/PrivateFrameworks/GameServicesFoundation.framework/Versions/A/gamed
  235 ??   0:01.90 /System/Library/CoreServices/cloudphotosd.app/Contents/MacOS/cloudphotosd
  236 ??   0:48.72 /usr/libexec/mediald
  239 ??   0:01.44 /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 ??   0:01.03 /System/Library/PrivateFrameworks/CallHistorySyncHelper.framework/Support/CallHistorysyncHelper
  243 ??   0:05.74 /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/CloudPhotosMessos.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/seed
  254 ??   0:24.31 /System/Library/PrivateFrameworks/CloudKitDaemon.framework/Support/cloudd
  258 ??   0:04.73 /System/Library/PrivateFrameworks/TelephonyUtilities.framework/callservicesd
  270 ??   0:00.40 /System/Library/PrivateFrameworks/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 ??   0:05.25 /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/Framework/CloudPhotoServicesConfiguration.framework/Versions/A/XPCServices/com.apple.CloudPhotosConfigur
ation.Connection.xpc/Contents/MacOS/com.apple.CloudPhotosConfiguration
  297 ??   0:06.88 /System/Library/PrivateFrameworks/CloudServices.framework/Resources/com.apple.akd
  302 ??   0:26.11 /System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
  303 ??   0:09.55 /System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer
```

```
ps -x
```

… 156 total at this moment

How does my laptop do this?

Imagine doing 156 assignments all at the same time!

Threads

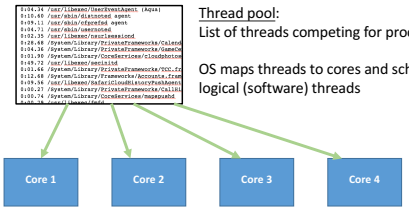
- Sequential flow of instructions that performs some task
  - Up to now we just called this a "program"
- Each thread has:
  - Dedicated PC (program counter)
  - Separate registers
  - Accesses the shared memory
- Each physical core provides one (or more)
  - *Hardware* threads that actively execute instructions
  - Each 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 1 ms)
2. Remove a software thread from a hardware thread by
   a) Interrupting its execution
   b) Saving its registers and PC to memory
3. Start executing a different software thread by
   a) Loading its previously saved registers into a hardware thread's registers
   b) Jumping to its saved PC

Example: Four Cores



Thread pool:
List of threads competing for processor

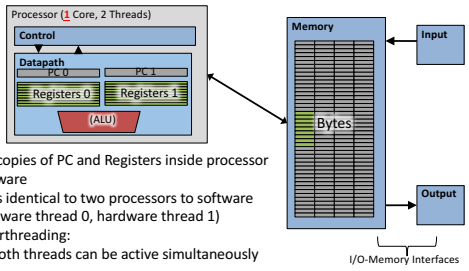
OS maps threads to cores and schedules logical (software) threads

Each "Core" actively runs one instruction stream at a time

Multithreading

- Typical scenario:
  - Active thread encounters cache miss
  - Active thread waits ~ 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 ≪ 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 identical to two processors to software (hardware thread 0, hardware thread 1)
- Hyperthreading:
  - Both threads can be active simultaneously

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

Randy'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:      4,194,304
```

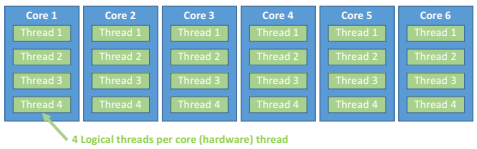
- 2 Cores
- 4 Threads total

Example: 6 Cores, 24 Logical Threads

```
0:14.34 /usr/libexec/InstEnvmtAgent (Aqua)
0:19.49 /usr/sbin/distnoted agent
0:09.11 /usr/sbin/distnoted agent
0:34.71 /usr/sbin/ntpd -g -c
0:02.35 /usr/libexec/nsurlsessiond
0:20.60 /System/Library/PrivateFrameworks/Calend
0:04.36 /System/Library/PrivateFrameworks/GameCe
0:01.90 /System/Library/CoreServices/cloudphotos
0:49.72 /usr/libexec/aeoinitd
0:01.66 /System/Library/PrivateFrameworks/TCC.fr
0:12.60 /System/Library/Frameworks/Accounts.fram
0:09.34 /usr/libexec/AssistiveUtilityStartPushAssist
0:00.27 /System/Library/PrivateFrameworks/CallHi
0:00.74 /System/Library/CoreServices/mapspushd
```

Thread pool:
List of threads competing for processor

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



Break!

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?

- 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/machine learning (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] + 10 *omp_get_thread_num();
        }

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

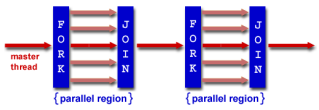
```
$ gcc-5 -fopenmp for.c;./a.out
thread 0, i =  0
thread 1, i =  3
thread 2, i =  6
thread 3, i =  8
thread 0, i =  1
thread 1, i =  4
thread 2, i =  7
thread 3, i =  9
thread 0, i =  2
thread 1, i =  5
01 02 03 14 15 16 27 28 39 40
```


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:**



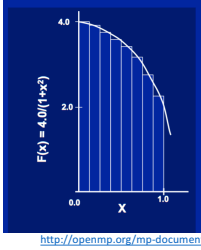
- OpenMP programs begin as single process (*master thread*)
  - Sequential execution
- When parallel region is encountered
  - Master thread "forks" into team of parallel threads
  - Executed simultaneously
  - At end of parallel region, parallel threads "join", leaving only master thread
- Process repeats 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 compete for hardware threads!
- Be "careful" (?) when timing results for Project 3!
  - 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

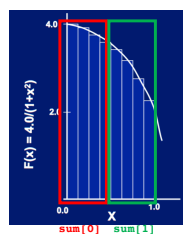
Parallelize (1) …

```
#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[1]** 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 = 10⁶`

```
#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?

Peer Instruction

What are the possible values of `*(x1)` after executing this code by two ***concurrent*** threads?

```
# *(x1) = 100
lw   x2,0(x1)
addi x2,x2,1
sw   x2,0(x1)
```

Answer	*(x1)
RED	100 or 101
GREEN	101
ORANGE	101 or 102
YELLOW	100 or 101 or 102

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, 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**

Administrivia

- Homework 4 (Caches, Floating Point) due **tomorrow** at 11:59pm
- Project 2-2 due Monday
  - Project Office hours that Monday will be well staffed!
  - Test your CPU thoroughly!
    - Write programs with Venus and load them into your circuit
- Project 3 will be released Monday night
  - A two-week performance project
  - Can earn extra credit from the performance contest (Project 5)
- Midterm scores will be released before Tuesday on Gradescope

Break!

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) ;

lock = 1;

// critical section

lock = 0;
```

= 0) ;

set,

ey got

and set the lock!

Try as you like, 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
- RISCV has variations of both, but for simplicity we will focus on the former

RISCV Atomic Memory Operations (AMOs)

- AMOs atomically perform an operation on an operand in memory and set the destination register to the original memory value
- R-Type Instruction Format: `Add`, `And`, `Or`, `Swap`, `Xor`, `Max`, `Max Unsigned`, `Min`, `Min Unsigned`

31–27	26	25	24–20	19–15	14–12	11–7	6–0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode
5	1	1	5	5	3	5	7
operation	ordering		src	addr	width	dest	AMO

Load from address in rs1 to "t"
rd = "t", i.e., the value in memory
Store at address in rs1 the calculation "t" <operation> rs2
aq and rl insure *in order* execution

```
amoadd.w rd,rs2,(rs1):
  t = M[x[rs1]];
  x[rd] = t;
  M[x[rs1]] = t + x[rs2]
```


RISCV Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is "set" if it is 1; it is "free" if it is 0 (it's initial value)

```
      li            t0, 1        # Get 1 to set lock
Try:  amoswap.w.aq  t1, t0, (a0) # t1 gets old lock value
                                 # while we set it to 1
      bnez          t1, Try      # if it was already 1, another
                                 # thread has the lock,
                                 # so we need to try again
          … critical section goes here …
      amoswap.w.rl  x0, x0, (a0) # store 0 in lock to release
```


Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

Fix (lock is at location (a0))

```
     li              t0, 1
Try  amoswap.w.aq    t1, t0, (a0)
     bnez            t1, Try
Locked:

     # critical section

Unlock:
     amoswap.w.rl    x0, x0, (a0)
```


Agenda

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

OpenMP Locks

```
#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 Section

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

void main () {
    const int NUM_THREADS = 1000;
    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;
    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