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
   - $< 5$GHz 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
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**

Harvest Parallelism & Achieve High Performance

Projects 3 and 5!
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

Processor “Core” 1

Control

Datapath

PC

Registers

(ALU)

Processor “Core” 2

Control

Datapath

PC

Registers

(ALU)

Memory

Bytes

Input

Processor 0 Memory Accesses

Processor 1 Memory Accesses

I/O-Memory Interfaces

Output

11/2/17

Fall 2017 - Lecture #19
Multiprocessor Execution Model

• Each processor (core) executes its own instructions

• **Separate** resources (not shared)
  – Datapath (PC, registers, ALU)
  – Highest level caches (e.g., 1\(^{st}\) and 2\(^{nd}\))

• **Shared** resources
  – Memory (DRAM)
  – Often 3\(^{rd}\) 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
<table>
<thead>
<tr>
<th></th>
<th>Pixel 2</th>
<th>iPhone 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>5.74 x 2.74 x 0.31 inches</td>
<td>5.45 x 2.65 x 0.29 inches</td>
</tr>
<tr>
<td>Weight</td>
<td>143 grams (5.04 ounces)</td>
<td>148 grams (5.22 ounces)</td>
</tr>
<tr>
<td>Screen</td>
<td>5-inch AMOLED display</td>
<td>4.7-inch Retina HD LCD-backlit widescreen</td>
</tr>
<tr>
<td>Resolution</td>
<td>1,920 x 1,080 pixels (441 ppi)</td>
<td>1,334 x 750 pixels (326 pixels-per-inch)</td>
</tr>
<tr>
<td>OS</td>
<td>Android 8.0</td>
<td>iOS 11</td>
</tr>
<tr>
<td>Storage</td>
<td>64GB, 128GB</td>
<td>64GB, 256GB</td>
</tr>
<tr>
<td>MicroSD card slot</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>NFC support</td>
<td>Yes</td>
<td>Yes, Apple Pay only</td>
</tr>
<tr>
<td>Processor</td>
<td>Snapdragon 835, with Adreno 540</td>
<td>A11 Bionic with 64-bit architecture, M11 motion coprocessor</td>
</tr>
<tr>
<td>RAM</td>
<td>4GB</td>
<td>2GB</td>
</tr>
<tr>
<td>Connectivity</td>
<td>GSM, CDMA, HSPA, EVDO, LTE, 802.11a/b/g/n/ac Wi-Fi</td>
<td>4G LTE, GSM, CDMA, HSPA+, 802.11a/b/g/n/ac Wi-Fi</td>
</tr>
<tr>
<td>Camera</td>
<td>12.2 MP rear, 8 MP HD front</td>
<td>12.2 MP rear, 8 MP HD front</td>
</tr>
<tr>
<td>Video</td>
<td>Up to 4K at 30fps, 1080p at 120fps, 720p at 240fps</td>
<td>Up to 4K at 60fps, 1080p at 240fps</td>
</tr>
</tbody>
</table>
Pixel 2 vs. iPhone 8

Qualcomm Technologies leading with 10nm

Kryo 280

A new generation of this microarchitecture, named Kryo 280, was announced along with the Snapdragon 835 chipset in November 2016. 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. The new core improves integer instructions per clock (+17%), while having much lower (-32%) performance at floating point math relative to the original Kryo.

Overview

- 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

<table>
<thead>
<tr>
<th>Adreno</th>
<th>ALUs</th>
<th>nm</th>
<th>MHz</th>
<th></th>
<th></th>
<th>GFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>540</td>
<td>256</td>
<td>10</td>
<td>710</td>
<td>?</td>
<td>?</td>
<td>587 (?)</td>
</tr>
</tbody>
</table>
Pixel 2 vs. iPhone 8

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. The A11 uses a new second-generation performance controller, which permits the A11 to use all six cores simultaneously, 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. Embedded in the A11 is the M11 motion coprocessor. The A11 includes a new image processor which supports computational photography functions such as lighting estimation, wide color capture, and advanced pixel processing. 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.

The A11 is manufactured by TSMC using a 10 nm FinFET process and contains 4.3 billion transistors on a die 87.66 mm² in size, 41% smaller than the A10. It is manufactured in a package on package (PoP) together with 2 GB of LPDDR4X memory in the iPhone 8 and 3 GB of LPDDR4X memory in the iPhone 8 Plus and iPhone X.
<table>
<thead>
<tr>
<th>Device</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Galaxy Note 8</td>
<td>6479</td>
</tr>
<tr>
<td>Samsung Exynos 8895 Octa @ 1.7 GHz</td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy S8</td>
<td>6437</td>
</tr>
<tr>
<td>Samsung Exynos 8895 Octa @ 1.7 GHz</td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy S8+</td>
<td>6406</td>
</tr>
<tr>
<td>Samsung Exynos 8895 Octa @ 1.7 GHz</td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy Note 8</td>
<td>6306</td>
</tr>
<tr>
<td>Qualcomm MSM8996 Snapdragon 855 @ 1.9 GHz</td>
<td></td>
</tr>
<tr>
<td>Xiaomi MI 6</td>
<td>6053</td>
</tr>
<tr>
<td>Qualcomm MSM8996 Snapdragon 855 @ 1.9 GHz</td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy S8+</td>
<td>6045</td>
</tr>
<tr>
<td>Qualcomm MSM8996 Snapdragon 855 @ 1.9 GHz</td>
<td></td>
</tr>
<tr>
<td>Huawei Honor V9</td>
<td>6024</td>
</tr>
<tr>
<td>HiSilicon Kirin 960 @ 1.8 GHz</td>
<td></td>
</tr>
<tr>
<td>Huawei P10</td>
<td>6000</td>
</tr>
<tr>
<td>HiSilicon Kirin 960 @ 1.8 GHz</td>
<td></td>
</tr>
<tr>
<td>Samsung Galaxy S8</td>
<td>5983</td>
</tr>
<tr>
<td>Qualcomm MSM8996 Snapdragon 855 @ 1.9 GHz</td>
<td></td>
</tr>
</tbody>
</table>

VS.

<table>
<thead>
<tr>
<th>Device</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPhone 8 Plus</td>
<td>10122</td>
</tr>
<tr>
<td>Apple A11 Bionic @ 2.4 GHz</td>
<td></td>
</tr>
<tr>
<td>iPhone 8</td>
<td>10030</td>
</tr>
<tr>
<td>Apple A11 Bionic @ 2.4 GHz</td>
<td></td>
</tr>
<tr>
<td>iPhone X</td>
<td>9939</td>
</tr>
<tr>
<td>Apple A11 Bionic @ 2.4 GHz</td>
<td></td>
</tr>
<tr>
<td>iPad Pro (10.5-inch)</td>
<td>9249</td>
</tr>
<tr>
<td>Apple A10X Fusion @ 2.3 GHz</td>
<td></td>
</tr>
<tr>
<td>iPad Pro (12.9-inch 2nd Generation)</td>
<td>9236</td>
</tr>
<tr>
<td>Apple A10X Fusion @ 2.3 GHz</td>
<td></td>
</tr>
<tr>
<td>iPhone 7</td>
<td>5753</td>
</tr>
<tr>
<td>Apple A10 Fusion @ 2.3 GHz</td>
<td></td>
</tr>
<tr>
<td>iPhone 7 Plus</td>
<td>5747</td>
</tr>
<tr>
<td>Apple A10 Fusion @ 2.3 GHz</td>
<td></td>
</tr>
<tr>
<td>iPad Pro (12.9-inch)</td>
<td>5017</td>
</tr>
<tr>
<td>Apple A9X @ 2.3 GHz</td>
<td></td>
</tr>
<tr>
<td>iPad Pro (9.7-inch)</td>
<td>4905</td>
</tr>
<tr>
<td>Apple A9X @ 2.3 GHz</td>
<td></td>
</tr>
</tbody>
</table>
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:
      o “Slow” memory shared by many “customers” (cores)
      o 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)*

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

**Potential Growth:**

- 2003 to 2005: *2X* / 2 years
- 2005 to 2007: *2X* / 2 years
- 2007 to 2009: *2X* / 2 years
- 2009 to 2011: *2X* / 2 years
- 2011 to 2013: *2X* / 2 years
- 2013 to 2015: *4X* / 4 years
- 2015 to 2017: *4X* / 4 years
- 2017 to 2019: *4X* / 4 years
- 2019 to 2021: *4X* / 4 years

**Formula:**

$$20^{1/12} = 1.28 \times 20 \text{ x in 12 years}$$

**Interpretation:**

- 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

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>220</td>
<td>??</td>
<td>0:04.34</td>
<td>/usr/libexec/UserEventAgent (Aqua)</td>
</tr>
<tr>
<td>222</td>
<td>??</td>
<td>0:10.60</td>
<td>/sbin/distnoted_agent</td>
</tr>
<tr>
<td>224</td>
<td>??</td>
<td>0:09.11</td>
<td>/sbin/cfprefsd_agent</td>
</tr>
<tr>
<td>229</td>
<td>??</td>
<td>0:04.71</td>
<td>/sbin/usernoted</td>
</tr>
<tr>
<td>230</td>
<td>??</td>
<td>0:02.35</td>
<td>/usr/libexec/nslsessiond</td>
</tr>
<tr>
<td>232</td>
<td>??</td>
<td>0:28.68</td>
<td>/System/Library/PrivateFrameworks/CalendarAgent.framework/Executables/CalendarAgent</td>
</tr>
<tr>
<td>234</td>
<td>??</td>
<td>0:04.36</td>
<td>/System/Library/PrivateFrameworks/GameCenterFoundation.framework/Versions/A/com.apple.XPCServices</td>
</tr>
<tr>
<td>235</td>
<td>??</td>
<td>0:01.90</td>
<td>/System/Library/CoreServices/cloudphotosd.app/Contents/MacOS/cloudphotosd</td>
</tr>
<tr>
<td>236</td>
<td>??</td>
<td>0:09.72</td>
<td>/usr/libexec/secinitd</td>
</tr>
<tr>
<td>239</td>
<td>??</td>
<td>0:01.66</td>
<td>/System/Library/PrivateFrameworks/TCC.framework/Resources/tccd</td>
</tr>
<tr>
<td>240</td>
<td>??</td>
<td>0:12.68</td>
<td>/System/Library/Frameworks/Accounts.framework/Versions/A/Support/accountsd</td>
</tr>
<tr>
<td>241</td>
<td>??</td>
<td>0:09.36</td>
<td>/usr/libexec/SafariCloudHistoryPushAgent</td>
</tr>
<tr>
<td>242</td>
<td>??</td>
<td>0:00.27</td>
<td>/System/Library/PrivateFrameworks/CallHistory.framework/Support/CallHistorySyncHelper</td>
</tr>
<tr>
<td>243</td>
<td>??</td>
<td>0:00.74</td>
<td>/System/Library/CoreServices/mapspushd</td>
</tr>
<tr>
<td>244</td>
<td>??</td>
<td>0:00.79</td>
<td>/usr/libexec/fmd</td>
</tr>
<tr>
<td>246</td>
<td>??</td>
<td>0:00.09</td>
<td>/System/Library/PrivateFrameworks/AskPermission.framework/Versions/A/Resources/askpermissiond</td>
</tr>
<tr>
<td>248</td>
<td>??</td>
<td>0:01.03</td>
<td>/System/Library/PrivateFrameworks/CloudDocsDaemon.framework/Versions/A/Support/bird</td>
</tr>
<tr>
<td>249</td>
<td>??</td>
<td>0:02.50</td>
<td>/System/Library/PrivateFrameworks/IDS.framework/identityservicesd.app/Contents/MacOS/identityservicesd</td>
</tr>
<tr>
<td>250</td>
<td>??</td>
<td>0:04.81</td>
<td>/usr/libexec/send</td>
</tr>
<tr>
<td>254</td>
<td>??</td>
<td>0:24.01</td>
<td>/System/Library/PrivateFrameworks/CloudKitDaemon.framework/Support/cloud</td>
</tr>
<tr>
<td>257</td>
<td>??</td>
<td>0:04.73</td>
<td>/System/Library/PrivateFrameworks/TelephonyUtilities.framework/callservicesd</td>
</tr>
<tr>
<td>267</td>
<td>??</td>
<td>0:02.15</td>
<td>/System/Library/CoreServices/AirPlayUIAgent.app/Contents/MacOS/AirPlayUIAgent --launchd</td>
</tr>
<tr>
<td>271</td>
<td>??</td>
<td>0:03.91</td>
<td>/usr/libexec/neuralstorage</td>
</tr>
<tr>
<td>274</td>
<td>??</td>
<td>0:00.90</td>
<td>/System/Library/PrivateFrameworks/CommerceKit.framework/Versions/A/Resources/storeaccountd</td>
</tr>
<tr>
<td>282</td>
<td>??</td>
<td>0:00.09</td>
<td>/sbin/pboard</td>
</tr>
<tr>
<td>283</td>
<td>??</td>
<td>0:00.90</td>
<td>/System/Library/PrivateFrameworks/InternetAccounts.framework/Versions/A/XPCServices/com.apple.internetaccounts.xpc/Contents/MacOS/com.apple.internetaccounts.xpc</td>
</tr>
<tr>
<td>285</td>
<td>??</td>
<td>0:04.72</td>
<td>/System/Library/Frameworks/ApplicationServices.framework/Frameworks/ATS.framework/Support/fontd</td>
</tr>
<tr>
<td>291</td>
<td>??</td>
<td>0:00.25</td>
<td>/System/Library/Frameworks/Security.framework/Versions/A/Resources/CloudKeychainProxy.bundle.bundle/Contents/MacOS/CloudKeychainProxy</td>
</tr>
<tr>
<td>292</td>
<td>??</td>
<td>0:09.64</td>
<td>/System/Library/CoreServices/CoreServicesUIAgent.app/Contents/MacOS/CoreServicesUIAgent</td>
</tr>
<tr>
<td>293</td>
<td>??</td>
<td>0:00.29</td>
<td>/System/Library/PrivateFrameworks/CloudPhotoServices.framework/Versions/A/Frameworks/CloudPhotoServicesConfiguration.framework/Versions/A/XPCServices/com.apple.CloudPhotosConfiguration.xpc/Contents/MacOS/com.apple.CloudPhotosConfiguration.xpc</td>
</tr>
<tr>
<td>297</td>
<td>??</td>
<td>0:00.84</td>
<td>/System/Library/PrivateFrameworks/CloudServices.framework/Resources/com.apple.sbd</td>
</tr>
<tr>
<td>302</td>
<td>??</td>
<td>0:26.11</td>
<td>/System/Library/CoreServices/Dock.app/Contents/MacOS/Dock</td>
</tr>
<tr>
<td>303</td>
<td>??</td>
<td>0:09.55</td>
<td>/System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer</td>
</tr>
</tbody>
</table>

```
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 $\sim$ 1000 cycles for data from DRAM
  – $\rightarrow$ 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)
  – $\rightarrow$ 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 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

Thread pool:
List of threads competing for processor

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

4 Logical threads per core (hardware) thread
Break!
Agenda

• MIMD - multiple programs simultaneously
• Threads
• **Parallel programming: OpenMP**
• Synchronization primitives
• Synchronization in OpenMP
• And, in Conclusion ...
## Languages Supporting Parallel Programming

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

*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?
      o General problem is very hard to solve
      o Present state: specialized solutions for specific cases
      o 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:

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

• Parallel Execution:

```java
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

```c
#include <omp.h>

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

```c
#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 = %d\n", 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, \texttt{#pragma}
  – Runtime Library Routines, \texttt{#include <omp.h>}
• \texttt{#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 $\pi$

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 $\Delta x$ and height $F(x_i)$ at the middle of interval $i$. 

http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
Sequential $\pi$

```c
#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 = %.12f\n", sum);
}
```

$\pi = 3.142425985001$

- Resembles $\pi$, but not very accurate
- Let’s increase `num_steps` and parallelize
Parallelize (1) …

```c
#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 thread needs access to the shared variable `sum`
- Code runs sequentially …
Parallelize (2) ...

1. Compute $\text{sum}[0]$ and $\text{sum}[1]$ in parallel

2. Compute $\text{sum} = \text{sum}[0] + \text{sum}[1]$ sequentially
#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);
#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);
        }
    }

    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: \texttt{num\_steps} = 10^6

```c
#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 \texttt{sum}?

```c
#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;
    #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?

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

<table>
<thead>
<tr>
<th>Answer</th>
<th>*(x1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED</td>
<td>100 or 101</td>
</tr>
<tr>
<td>GREEN</td>
<td>101</td>
</tr>
<tr>
<td>ORANGE</td>
<td>101 or 102</td>
</tr>
<tr>
<td>YELLOW</td>
<td>100 or 101 or 102</td>
</tr>
</tbody>
</table>
What’s Going On?

```c
#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 ("\npi = %6.12f\n", pi);
}
```

• Operation is really
  \[ \text{pi} = \text{pi} + \text{sum[\text{id}]} \]
• What if >1 threads reads current (same) value of \text{pi}, computes the sum, stores the result back to \text{pi}?
• Each processor reads same intermediate value of \text{pi}!
• Result depends on who gets there when
  • A “race” \(\rightarrow\) 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 gets 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;

Thread 2

while (lock != 0) ;
lock = 1;

// critical section

lock = 0;

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
RISC V 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

<table>
<thead>
<tr>
<th>funct5</th>
<th>aq</th>
<th>rl</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AMO</td>
</tr>
</tbody>
</table>

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**

```plaintext
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

**Fix (lock is at location (a0))**

```plaintext
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

```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 $pragma$ 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
#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