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

Improving Performance

1. Increase clock rate $f_s$ - Reached practical maximum for today's technology
   - < 6 GHz for general purpose computers
2. Lower CPI (cycles per instruction)
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
4. Do all of the above:
   - High $f_s$, SIMD, multiple parallel tasks

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

Example: CPU with Two Cores

Parallel Computer Architectures
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

- AM2 (Pentium II core)
- Intel Pentium 4
- AMD Athlon 64

Pixel 2 vs. iPhone 8

<table>
<thead>
<tr>
<th>Design</th>
<th>Pixel 2</th>
<th>iPhone 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Qualcomm Snapdragon 660 with Android Oreo</td>
<td>Apple A11 Bionic with iOS 11</td>
</tr>
<tr>
<td>Screen</td>
<td>5.7&quot; 2.74 x 5.73 inches</td>
<td>4.7&quot; 2.56 x 5.29 inches</td>
</tr>
<tr>
<td>Size</td>
<td>5.6&quot; 2.68 x 5.29 inches</td>
<td>4.7&quot; 2.56 x 5.29 inches</td>
</tr>
<tr>
<td>Resolution</td>
<td>1920 x 1080 pixels (411 dpi)</td>
<td>1334 x 750 pixels (326 pixels per inch)</td>
</tr>
<tr>
<td>Camera</td>
<td>12.2 MP max, x 1.8 MP front</td>
<td>12.2 MP max, x 1.8 MP front</td>
</tr>
<tr>
<td>Connectivity</td>
<td>4G LTE, GSM, WCDMA, HSPA, BLU 11.6Gbps (Wi-Fi)</td>
<td>N/A</td>
</tr>
<tr>
<td>Storage</td>
<td>64GB, 128GB, 32GB</td>
<td>64GB, 128GB, 256GB</td>
</tr>
<tr>
<td>Battery</td>
<td>Up to 88% of 3900mAh</td>
<td>Up to 88% of 2105mAh</td>
</tr>
</tbody>
</table>

Pixel vs. iPhone 8

| Processor | Qualcomm Snapdragon 835 with Android Oreo | Apple A11 Bionic with iOS 11 |
| Device | Samsung Galaxy S8 | Apple iPhone 8 |
| Benchmark | Samsung/Qualcomm S8 Plus (6.2" OLED) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Samsung Galaxy S8 (5.8" QHD) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 7 Plus (5.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 7 (4.7" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 6S (4.7" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 6 (4.7" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 5s (4.0" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 5 (3.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 4s (3.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 4 (3.0" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 3GS (3.0" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 3G (3.0" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 2G (2.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone 1G (2.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPhone (2.5" Retina) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Touch (3rd generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Touch (2nd generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Classic (4th generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Classic (3rd generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Classic (2nd generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Classic (1st generation) | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Classic | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Video | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod Photo | Apple iPhone 8 (4.7" Retina) |
| Benchmark | Apple iPod | Apple iPhone 8 (4.7" Retina) |

<table>
<thead>
<tr>
<th>Device</th>
<th>Single Case</th>
<th>Multi Case</th>
<th>Headset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung Galaxy S8</td>
<td>5897</td>
<td>6240</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 8</td>
<td>17292</td>
<td>17292</td>
<td>159</td>
</tr>
<tr>
<td>Samsung Galaxy S8 Plus</td>
<td>5427</td>
<td>6388</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 8 Plus</td>
<td>7353</td>
<td>7353</td>
<td>159</td>
</tr>
<tr>
<td>Samsung Galaxy S8 (6.2&quot; OLED)</td>
<td>6122</td>
<td>6122</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 8 (4.7&quot; Retina)</td>
<td>4576</td>
<td>4576</td>
<td>159</td>
</tr>
<tr>
<td>Samsung Galaxy S8 (5.8&quot; QHD)</td>
<td>6088</td>
<td>6088</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 8 (4.7&quot; Retina)</td>
<td>4576</td>
<td>4576</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 7 Plus (5.5&quot; Retina)</td>
<td>5153</td>
<td>5153</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 7 (5.5&quot; Retina)</td>
<td>5153</td>
<td>5153</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 6S (5.5&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 6 (5.5&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 5s (4.7&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 5 (4.0&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 4s (3.5&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 4 (3.0&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 3GS (3.0&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPhone 3G (3.0&quot; Retina)</td>
<td>5931</td>
<td>5931</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Classic (4th generation)</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Classic (3rd generation)</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Classic (2nd generation)</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Classic (1st generation)</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Classic</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Video</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod Photo</td>
<td>4836</td>
<td>4836</td>
<td>159</td>
</tr>
<tr>
<td>Apple iPod</td>
<td>4836</td>
<td>4836</td>
<td>159</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:
  - Simplicity in communication in programs via shared variables
- Drawbacks:
  - Does not scale well
    - "Slow" memory shared by many "subcores" (cores)
  - May become bottleneck (Kilmari’s Law)
- Two ways to use a multiprocessor:
  - Job-level parallelism
    - Processors work on unrelated problems
  - Partition work of single task between several cores
    - E.g., each core 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 lat /Core</th>
<th>Core - SIMD lat</th>
<th>Result, e.g.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>MIMD</td>
<td>128 265</td>
<td></td>
<td>4 MIMD</td>
</tr>
<tr>
<td>2005</td>
<td>2x/ 4</td>
<td>128 512</td>
<td></td>
<td>8 SIMD</td>
</tr>
<tr>
<td>2007</td>
<td>6yrs 6</td>
<td>128 768</td>
<td></td>
<td>12 SIMD</td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>128 1024</td>
<td></td>
<td>16 SIMD</td>
</tr>
<tr>
<td>2010</td>
<td>10</td>
<td>256 2560</td>
<td></td>
<td>40 SIMD</td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>256 3072</td>
<td></td>
<td>48 SIMD</td>
</tr>
<tr>
<td>2015</td>
<td>2.5x 14</td>
<td>512 7168</td>
<td></td>
<td>112 SIMD</td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>512 8192</td>
<td></td>
<td>128 SIMD</td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>1024 18432</td>
<td></td>
<td>288 SIMD</td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>1024 20480</td>
<td></td>
<td>320 SIMD</td>
</tr>
</tbody>
</table>

2x or 1.28x → 20% per year or 2x every 5 years!
If 4x 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

- 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

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 datapaths, 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

Thread 1
Core 2
Thread 2
Thread 3
Thread 4
Core 6
Thread 1
Core 4
Thread 2
Thread 3
Thread 4
Core 5
Thread 1
Core 3
Thread 2
Thread 3
Thread 4
Core 1
Thread 1
Core 1
Thread 2
Thread 3
Thread 4

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>Language</th>
<th>Concurrent Pascal</th>
<th>Java</th>
<th>Orc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada</td>
<td>Concurrent ML</td>
<td>C</td>
<td>O2</td>
</tr>
<tr>
<td>Aflax</td>
<td>Concurrent Haskell</td>
<td>Java</td>
<td>Pict</td>
</tr>
<tr>
<td>Aler</td>
<td>Curry</td>
<td>Joule</td>
<td>Rea</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>Ascum</td>
<td>Effel</td>
<td>Limbo</td>
<td>SISAL</td>
</tr>
<tr>
<td>Chapel</td>
<td>Erlang</td>
<td>Linda</td>
<td>SRL</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</td>
<td>X</td>
</tr>
</tbody>
</table>

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
• 6.1C
  • 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++) {
  ... 
  }
  ```

OpenMP Example

```c
#include <omp.h>

int main() {
  int all[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  int A[5] = {1, 2, 3, 4, 5};
  int B[6] = {1, 2, 3, 4, 5, 6};
  int C[7] = {1, 2, 3, 4, 5, 6, 7};
  #pragma omp parallel for
  for (int i=0; i<100; i++) {
    printf("%d\n", all[i]);
  }
}
```

OpenMP

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

OpenMP Programming Model

• Fork - Join Model:
  ```
  master thread
  (parallel region)
  ```
• 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
• All 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!
  – SAM?
  – Job queue?
Numerical integration

Mathematically, we know that

$$\int \left( \frac{1}{x^2} \right) dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum \frac{\text{width} \times \text{height}}{n}$$

Where each rectangle has width $\Delta x$ and height $f(x)$ at the middle of each interval.

Example 2: Computing $\pi$

Sequential $\pi$

```c
#include <stdio.h>

void main() {
    const long num_steps = 100;
    double step = 1.0/(double)num_steps;
    double sum = 0.0;
    double x = 1.0;
    for (int i = 1; i <= num_steps; i++) {
        double dx = step * i;
        sum += dx * (1.0/x);
    }
    printf("pi = \%.15f", sum);
}
```

- $\pi \approx 3.141592653589793\) ≤ 0.01
- Let's increase $\text{num\_steps}$ and parallelize

Parallelize (1) ...

```c
#include <stdio.h>

void main() {
    const long num_steps = 100;
    double step = 1.0/(double)num_steps;
    double sum = 0.0;
    double x = 1.0;
    for (int i = 1; i <= num_steps; i++) {
        double dx = step * i;
        sum += dx * (1.0/x);
    }
    printf("pi = \%.15f", sum);
}
```

Problem: each thread needs access to the shared variable `$sum$`
- Code runs sequentially...

Parallelize (2) ...

```c
#include <stdio.h>

void main() {
    const long num_steps = 100;
    double step = 1.0/(double)num_steps;
    double sum0[NUM_THREADS];
    double sum1[NUM_THREADS];
    for (int i = 0; i < num_steps; i++) {
        double dx = step * i;
        sum0[i] = dx * (1.0/x);
        sum1[i] = dx * (1.0/x);
    }
    double sum0 = 0;
    double sum1 = 0;
    for (int i = 0; i < num_steps; i++) {
        sum0 += sum0[i];
        sum1 += sum1[i];
    }
    printf("pi = %.15f", sum0);
}
```

1. Compute `$\text{sum[0]}$` and `$\text{sum[1]}$` sequential
2. Compute `$\text{sum} = \text{sum[0]} + \text{sum[1]}$` sequentially

Parallel $\pi$

Trial Run

```c
#include <stdio.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 100;
    double step = 1.0/(double)num_steps;
    double sum[NUM_THREADS];
    double sum0[NUM_THREADS];
    double sum1[NUM_THREADS];
    for (int i = 0; i < num_steps; i++) {
        double dx = step * i;
        sum0[i] = dx * (1.0/x);
        sum1[i] = dx * (1.0/x);
    }
    double sum0 = 0;
    double sum1 = 0;
    for (int i = 0; i < num_steps; i++) {
        sum0 += sum0[i];
        sum1 += sum1[i];
    }
    printf("pi = %.15f", sum0);
}
```

- $\pi \approx 3.141592653589793\) ≤ 0.01
- Let's increase `$\text{num\_steps}$` and parallelize

```c
#include <stdio.h>

void main () {
    const int NUM_THREADS = 4;
    const long num_steps = 100;
    double step = 1.0/(double)num_steps;
    double sum[NUM_THREADS];
    double sum0[NUM_THREADS];
    double sum1[NUM_THREADS];
    for (int i = 0; i < num_steps; i++) {
        double dx = step * i;
        sum0[i] = dx * (1.0/x);
        sum1[i] = dx * (1.0/x);
    }
    double sum0 = 0;
    double sum1 = 0;
    for (int i = 0; i < num_steps; i++) {
        sum0 += sum0[i];
        sum1 += sum1[i];
    }
    printf("pi = %.15f", sum0);
}
```

- $\pi \approx 3.141592653589793\) ≤ 0.01
Scale up: num_steps = $10^6$

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

void main () {
    const long num_steps = 1000000;
    double step = 1.0/(double)(num_steps);
    for (int id = 0; id < num_steps; ++id) {
        double x = 1.0;
        for (int i = 0; i < 100; ++i) {
            x = x * (1.0 - step);
        }
        double pi = 4.0 * atan(x);
        printf("pi = %f\n", pi);
    }
}
```

You verify how many digits are correct ...

Can We Parallelize Computing sum?

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

void main () {
    const int NUM_THREADS = 1000000;
    double sum = 0.0;
    double step = 1.0/(double)(NUM_THREADS);
    for (int id = 0; id < NUM_THREADS; ++id) {
        double pi = 0.0;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
            int id = omp_get_thread_num();
            for (int i = 0; i < 100; ++i) {
                double x = 1.0;
                for (int j = 0; j < 100; ++j) {
                    x = x * (1.0 - step);
                }
                x = 4.0 * atan(x);
                pi += x[i];
            }
        }
        printf("pi = %f\n", pi);
    }
}
```

Peer Instruction

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

<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>

Always looking for ways to beat Amdahl's Law ...

What's Going On?

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

void main () {
    const int NUM_THREADS = 1000000;
    double sum = 0.0;
    double step = 1.0/(double)(NUM_THREADS);
    for (int id = 0; id < NUM_THREADS; ++id) {
        double pi = 0.0;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
            int id = omp_get_thread_num();
            for (int i = 0; i < 100; ++i) {
                double x = 1.0;
                for (int j = 0; j < 100; ++j) {
                    x = x * (1.0 - step);
                }
                x = 4.0 * atan(x);
                pi += x[i];
            }
        }
        printf("pi = %f\n", pi);
    }
}
```

Surrogation inside parallel section
- Insignificant speedup in this example, but ...
- \(\pi = 3.14159265390\)
- Wrong! And value changes between runs?!
- What's going on?

What's Going On?

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

void main () {
    const int NUM_THREADS = 1000000;
    double sum = 0.0;
    double step = 1.0/(double)(NUM_THREADS);
    for (int id = 0; id < NUM_THREADS; ++id) {
        double pi = 0.0;
        omp_set_num_threads(NUM_THREADS);
        #pragma omp parallel
        {
            int id = omp_get_thread_num();
            for (int i = 0; i < 100; ++i) {
                double x = 1.0;
                for (int j = 0; j < 100; ++j) {
                    x = x * (1.0 - step);
                }
                x = 4.0 * atan(x);
                pi += x[i];
            }
        }
        printf("pi = %f\n", pi);
    }
}
```

- Operation is really \(\pi = \pi + \text{sum}[id]\)
- What if \(1\) thread reads current (wrong) 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!
- 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

```c
// 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
  ```c
  while (lock != 0) ;
  lock = 1;
  // critical section
  lock = 0;
  ```

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, Min, Min Unsigned.

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

RISC V 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
bez 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
```

OpenMP Locks

```
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
bez 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
```

Synchronization in OpenMP

- Typically used in libraries of higher level parallel programming constructs
- E.g. OpenMP offers $pragmas$ for common cases:
  - critical
  - atomic
  - barrier
  - ordered
- OpenMP offers more many more features
  - See online documentation
  - Or tutorial at:

Agenda

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

Lock Synchronization

```
while (lock != 0) ;
lock = 1;
// critical section
lock = 0;
```

Fix (lock is at location (a0))

```
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
bez 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
```
OpenMP Critical Section

```c
#include <omp.h>

main () { 
  // critical section
  double double_val = 9999;
  // parallel code

  // barrier synchronization
  int id = omp_get_thread_num();
  for (int i = 0; i < num_threads; i++) {
    if (i == id) {
      // critical section
    }
  }
}
```

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
    - Multithreaded
    - 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