Unit 18

Performance & Parallelism
Review of Software Execution (1)

- Recall:
  1. High level code is compiled to assembly/machine code instructions
  2. When executed, CPU uses PC register to know what instruction to fetch/execute next
  3. PC is incremented and used to fetch the next instruction
  4. Registers are used as temp. storage of variables
  5. Variables "live" in memory & must be updated

Note this `init()` function initializes a million element array with values 0 to 999999

Next slide shows the execution of the 'init' for loop at several points in time (with some details omitted).

```c
#define MAX 1000000;
int data[MAX];

void init()
{
    int i;
    for(i=0; i < MAX; i++)
    {
        data[i] = i;
    }
}

int main()
{
    init();
    /* More work */
    return 0;
}
```
Review of Software Execution (2)

Time 1: Store
- CPU
- PC 800a5
- ALU
- Regs
- s2: 0

Bus (Addr, Data, Control)

Memory
init:
---
L1: // i <=> s2
store s2,(sf)
add s2,1
compare s2,MAX
jump NZ,L1

Time 2: Increment i
- CPU
- PC 800a6
- ALU
- Regs
- s2: 1

Bus (Addr, Data, Control)

Memory
init:
---
L1: // i <=> s2
store s2,(sf)
add s2,1
compare s2,MAX
jump NZ,L1

Time 3: Loop & repeat
- CPU
- PC 800a5
- ALU
- Regs
- s2: 1

Bus (Addr, Data, Control)

Memory
init:
---
L1: // i <=> s2
store s2,(sf)
add s2,1
compare s2,MAX
jump NZ,L1

Time 4
- CPU
- PC 800a6
- ALU
- Regs
- s2: 2

Bus (Addr, Data, Control)

Memory
init:
---
L1: // i <=> s2
store s2,(sf)
add s2,1
compare s2,MAX
jump NZ,L1

Some details have been left out (how does sf get initialized, incremented, etc.)
Improving Performance

• We want to improve the performance of our computation

• Question: What are we referring to when we say "performance"?
  – Speed
  – Energy consumption
  – Cost?

• We will primarily consider speed in this discussion
How Do We Measure Speed

- **Fundamental Measurement:** TIME
  - Absolute time from start to finish
  - To compare two alternative systems (HW + SW) and their performance, start a timer when you begin a task and stop it when the task ends
  - Do this for both systems and compare the resulting times
- We call this the *latency* of the system and it works great from the perspective of the single, overall task
  - If system A completes the task in 2 seconds and system B requires 3 seconds, then system A is clearly superior
- But when we dig deeper and realize that the single, overall task is likely made of many small tasks, we can consider more than just latency
Performance Depends on View Point?!

• What's faster to get from point A to point B?
  – A 747 Jumbo Airliner
  – An F-22 fighter jet

• If only 1 person needs to get from point A to point B, then the F-22
  – This is known as **latency [units of seconds]**
  – Time from the start of an operation until it completes

• If 200 people need to get from point A to point B, the 747 looks much better
  – This is known as **throughput [jobs/second]**

• The **overall** execution time (latency) may best be improved by increasing throughput and not the latency of individual tasks
Improving Latency and Throughput

CACHING AND PIPELINING
Hardware Techniques

- We can add hardware or reorganize our hardware to improve throughput and latency of individual tasks in an effort to reduce the total latency (time) to finish the overall task.

- We will look at two examples:
  - Caching: Improves latency
  - Pipelining: Improves throughput
Caching

- **Cache (def.)** – "to store away in hiding or for future use"

- Primary idea
  - The first time you access or use something you expend the full amount of time to get it
  - However, store it someplace (i.e. in a cache) you can get it more quickly the next time you need it
  - The next time you need something check if it is in the cache first
  - If it is in the cache, you can get it quickly; else go get it expending the full amount of time (but then save it in the cache)

- Examples:
  - Web-browser
  - Checking out a book from the library
  - Your refrigerator
Cache Overview

• Remember what register are used for?
  – Quick access to copies of data
  – Only a few (32 or 64) so that we can access really quickly
  – Controlled by the software/compiler

• Cache memory is a small-ish, (kilobytes to a few megabytes) "fast" memory usually built onto the processor chip

• Will hold copies of the latest data & instructions accessed by the processor

• Managed by the HW
  – Transparent to the software
Cache Operation (1)

- When processor wants data or instructions it always checks in the cache first
- If it is there, fast access
- If not, get it from memory
- Memory will also supply surrounding data since it is likely to be needed soon
  - Why?
  - Things like arrays & code (instructions) are commonly accessed sequentially
Cache Operation (2)

- When processor asks for the data again or for the next data value in the array (or instruction of the code) the cache will likely have it.
- Questions?

Main point: Caching reduces the latency of memory accesses which improves overall program performance.
Memory Hierarchy & Caching

- Use several levels of faster and faster memory to hide latency of larger levels

Unit of Transfer:
- Registers: 8-64 bits
- L1 Cache: ~1 ns
- L2 Cache: ~10 ns
- Main Memory: ~100 ns

Unit of Transfer:
- 8-64 bytes
Pipelining

- We'll now look at a hardware technique called **pipelining** to improve **throughput**
- The key idea is to **overlap** the processing of multiple "items" (either data or instructions)
Example

- Suppose you are asked to build dedicated hardware to perform some operation on all 100 elements of some arrays
- Suppose the operation \((A[i] + B[i]) / 4\) takes 10 ns to perform
- How long would it take to process the entire arrays: **1000 ns**
  - Can we improve?

\[
\text{for}(i=0; \ i < 100; \ i++) \\
\text{C}[i] = (A[i] + B[i]) / 4;
\]

Clock Freq. = \(1/10\text{ns} = 100 \text{ MHz}\) (longest path from register to register)
**Pipelining Example**

- Pipelining refers to insertion of registers to split combinational logic into smaller stages that can be overlapped in time (i.e. create an assembly line)

**Stage 1**

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A[0] + B[0]</td>
</tr>
</tbody>
</table>

**Stage 2**

<table>
<thead>
<tr>
<th>Clock Cycle</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(A[0] + B[0]) / 4</td>
</tr>
</tbody>
</table>

---

**for(i=0; i < 100; i++)**

\[ C[i] = \frac{A[i] + B[i]}{4}; \]

- Time for 0th elements to complete: 10ns
- Time between each of the remaining 99 element completing: 5ns
- Total: \( 10 + 99 \times 5 = 505 \text{ns} \)

Define:

\[ \text{speedup} = \frac{\text{Original time}}{\text{Improved Time}} \]

\[ \text{speedup} = \frac{1000\text{ns}}{505\text{ns}} \approx 2x \]

Clock freq: \( \frac{1}{5\text{ns}} = 200\text{MHz} \)
Need for Registers

- Provides separation between combinational functions
  - Without registers, fast signals could “catch-up” to data values in the next operation stage

Performing an operation yields signals with different paths and delays.

We don't want signals from two different data values mixing. Therefore we must collect and synchronize the values from the previous operation before passing them on to the next.
Pipelining Example

- By adding more pipelined stages we can improve throughput
- Have we affected the latency of processing individual elements? No!!!
- Questions/Issues?
  - Balancing stage delays
  - Overhead of registers (Not free to split stages)
    - This limits how much we can split our logic

```c
for(i=0; i < 100; i++)
    C[i] = (A[i] + B[i]) / 4;
```

- Time for 0th elements to complete: 10 ns
- Time between each of the remaining 99 element completing: 2.5 ns
- Total: 257.5 ns

\[
\text{speedup} = \frac{1000\text{ns}}{257.5\text{ns}} \approx 4x
\]
Non-Pipelined Processors

• Currently we know our processors execute software 1 instruction at a time

• 3 steps/stages of work for each instruction are:
  – Fetch
  – Decode
  – Execute
Pipelined Processors

- By breaking our processor hardware for instruction execution into stages we can overlap these stages of work.
- Latency for a single instruction is the same.
- Overall throughput, and thus total latency, are greatly improved.

Diagram:

- `FDE` for `instruc. i`
- `FDE` for `instruc. i+1`
- `FDE` for `instruc. i+2`
- `FDE` for `instruc. i+3`
More and More Stages

- We can break the basic stages of work into substages to get better performance.
- In doing so our clock period goes down; frequency goes up.
- All kinds of interesting issues come up though when we overlap instructions and our discussed in future CENG courses.

Clock freq. = 1/10ns = 100MHz

Clock freq. = 1/5ns = 200MHz
Summary

• By investing extra hardware we can improve the overall latency of computation

• Measures of performance:
  – Latency is start to finish time
  – Throughput is tasks completed per unit time (measure of parallelism)

• Caching reduces latency by holding data we will use in the future in quickly accessible memory

• Pipelining improves throughput by overlapping processing of multiple items (i.e. an assembly line)
PARALLELISM
Recall Where We Started

- Recall:
  1. High level code is compiled to assembly/machine code instructions
  2. When executed, CPU uses PC register to know what instruction to fetch/execute next
  3. PC is incremented and used to fetch the next instruction
  4. Registers are used as temp. storage of variables
  5. Variables "live" in memory & must be updated

```c
const int MAX = 1000000;
int data[MAX];

void init()
{
    int i;
    for(i=0; i < MAX; i++){
        data[i] = i;
    }
}

int main()
{
    init();
    /* More work */
    return 0;
}
```
Diminishing Returns

- At some point our ability to speed up the hardware diminishes.
- Additionally, the faster we run our hardware the more power/energy we burn.
  - \( \text{Power} \propto \text{frequency}^3 \)
- Suppose we have 1 processor running at 1GHz using 10 Watts.
  - It can do 1 billion operations/second.
- Suppose we improve (pipeline) it to run at 2GHz.
  - It can do 2 billion operations/second but uses 80 Watts.
- Too much power means too much heat.
  - This can damage the chip if not managed carefully.
- Suppose we have 2 processor cores running at 1 GHz (10W each).
  - We can still do 2 billion operations/second (1 billion each).
  - But now we only require 20 Watts.
- To continue improving performance at reasonable power levels the trend is to parallel processing (multiple processor cores per chip).
Improving Performance

• Processors can only do one thing at a time
  – Initializing 1 million elements is a lot of "one" things

• Suppose you volunteer for USC events and have to place a give away item below every seat in the Galen Center
  – What could you do to get that task done faster?
  – Enlist your friends and each take a portion of the seats

• By enlisting the help of many processors we can often finish tasks faster
  – We can split the array into portions and each processor work on one portion

```c
const int MAX = 1000000;
int data[MAX];

void init()
{
    int i;
    for(i=0; i < MAX; i++)
    {
        data[i] = i;
    }
}

int main()
{
    init();
    /* More work */
    return 0;
}
```
Multi-Core Processors

- Multi-core processors (aka chip multiprocessors) duplicate the logic of a single processor and place several copies on the same chip.
- Each processor can be executing a separate program simultaneously.
- This improves throughput but NOT the latency of a single program.

```c
int main()
{
    int x = 0;
    for(...)
    /* More work */
        return 0;
}
```

`prog1.cpp`

```c
int main()
{
    int z = 0;
    while(cin >> z)
        /* More work */
        return 0;
}
```

`prog2.cpp`
Multithreaded Programs

• OR rather than executing different programs each core can be working collaboratively on the same problem
  – This can help us improve the latency of single program
• This requires us to break our work into tasks (aka threads)

```c
const int MAX = 1000000;
int data[MAX];

void init_t1()
{
    for(int i=0; i < MAX/2; i++)
        data[i] = i;
}

void init_t2()
{
    for(int i=MAX/2; i < MAX; i++)
        data[i] = i;
}

int main()
{
    create_parallel_task( init_t1 );
    create_parallel_task( init_t2 );
    wait_for_tasks_to_finish();
    /* more work */
}
```
Kinds of Parallelism

• To improve latency we want to start thinking how we can decompose our single program into parallel subtasks

• Task parallelism
  – Different tasks that don't need to be executed in a particular order
  – Example 1: Doing EE 109 homework vs. doing MATH homework
  – Example 2: Video game (AI engine, User Input/Output, Graphics and animation rendering)

• Data parallelism
  – Perform similar/same operations on many independent data elements (Same code operations running on different portions of data)
  – Usually taking iterations of a 'for' loop and splitting them up between tasks
  – **We will focus on data parallelism in this class**

```c
const int MAX = 1000000;
int data[MAX];

void init()
{
    int i;
    for(i=0; i < MAX; i++)
    {
        data[i] = i;
    }
}

int main()
{
    init();
    /* More work */
    return 0;
}
```

This would be an example of data parallelism
Ideal Parallel Performance Gain

• Define:

Parallel speedup = Sequential Exec. time / Parallel Exec. time

  – If a sequential program runs in 24 sec. and we parallelize it to run in 3 sec. our speedup is 24/3 = 8 times faster

• Ideal parallel speedup with N processor cores is N

  – Known as linear speedup
  – It is often hard to achieve linear speedup
  – Why?

    • Communication overhead (can you write a term paper n times faster given n teammates)?
Task 1: Initialize an array

- Suppose you are given a large array of integers to initialize, each thread could initialize a portion sum, how could you parallelize it?
  - Initialization is trivial but as a similar example, consider determining if integer i is a prime number (which takes a "reasonable" amount of work) for all integers up to MAX
- Is any communication needed? No!

```c
const int MAX = 100000;
int data[MAX];

void init_all()
{
    for(int i=0; i < MAX; i++){
        /* init element i */
    }
}
int main()
{
    init_all();
    /* More work */
    return 0;
}
```
Task 2: Sum a Large Array of Numbers

- Suppose you are given a large array of integers to sum, how could you parallelize it?
- What communication would need to take place between tasks?
  - Adding to the sum variable

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int start, int n)
{
    for(int i=start; i<start+n; i++){
        sum += data[i];
    }
}

int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```
Shared Variables

• Recall all variables live in memory
• To update a variable (e.g. `sum = sum + data[i]`) the CPU must:
  – Read the variable from memory into a register
  – Modify the value (perform the addition)
  – Write (save) the new value back to memory

• Note: Even with 2 processors, the bus limits parallelism (one access at a time)

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int start, int n)
{
    for(int i=start; i < start+n; i++){
        sum += data[i]; // RMW cycle
    }
}
int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```

These 3 steps, which happen sequentially, are known as a Read-Modify-Write (RMW) cycle.
Order of Shared Variable Access

- The **read-modify-write** cycle of parallel threads can be performed in an arbitrary order (i.e. interleaved in any order)
  - It's possible that two threads will both read the old value of sum and each add their respective data element to that old sum; upon writeback we will not get the correct sum

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int start, int n)
{
    for(int i=start; i < start+n; i++){
        sum += data[i]; // RMW cycle
    }
}

int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```

**Potential Ordering Timeline**

- **Thread 1**
  - Read sum (sum = 0)
  - Add data[0]
  - Write sum (sum = 1)
- **Thread 2**
  - Read sum (sum = 0)
  - Add data[4]
  - Write sum (sum = 5) → **WRONG**
Synchronized (Atomic) Access

- For variables that are shared and updated by multiple threads we need to synchronize the access
- Each thread should do its full Read-Modify-Write cycle uninterrupted (atomically) to ensure two threads don't read the same value of sum
  - Just as an atom cannot be "split" apart, we don't want the RMW cycle to be split up

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int start, int n)
{
    for(int i=start; i < start+n; i++){
        ATOMIC(sum += data[i]);
    }
}

int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```
Synchronized Access & Performance

- Suppose we do enforce synchronized (atomic) access to the sum variable, what is the effect on performance?
  - We lose most of the benefit of parallelism
  - To ensure correctness we update sum one transaction at a time
  - We say that access has been "serialized" (1 at a time = same as 1 thread = BAD!)

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int s, int n)
{
    for(int i=s; i < s+n; i++){
        ATOMIC(sum += data[i]);
    }
}

int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```
Private Variables

- Do we have the same problem of unsynchronized access to the loop counter, \( i \)?
- No, variables declared inside the thread are **private**
  - **Private** = Each thread has their own version (Thread 0 uses \( i = 0 \ldots 499,999 \) while Thread 1 uses \( i = 500,000 \) to \( 999,999 \))
  - Private variables can be accessed in parallel (by bringing them into a register or cache memory local to each core)

```c
const int MAX = 1000000;
int data[MAX];
int sum = 0;

void do_sum(int s, int n)
{
    for(int i=s; i < s+n; i++){
        ATOMIC( sum += data[i] );
    }
}

int main()
{
    create_thread(do_sum(0,MAX/2));
    create_thread(do_sum(MAX/2,MAX));
    return 0;
}
```
Improving Performance with Private Variables

• Can we apply the idea of private variables to the shared sum variable
  – Yes!! Use a private `lsum` ("local sum") to sum all the elements a thread is responsible for
  – Access to `lsum` need not be atomic since each thread has their own

• We need only combine T (# of threads) local sums as threads finish

```cpp
const int MAX = 1000000;
int data[MAX];

int do_sum(int s, int n)
{
    int lsum = 0; // private=per thread
    for(int i=s; i < s+n; i++){
        lsum += data[i]; // does this need
                          // to be atomic? No!
    }
    return lsum;
}

int main()
{
    const int T = /* # of threads */;
    int n = MAX/T;
    int sum = 0;
    for(int i=0; i < T; i++)
    {
        create_thread(do_sum(i*n, n));
    }
    /* Let parallel work happen */
    for(int i=0; i < T; i++)
    {
        // get returned lsums from threads
        sum += wait_for_thread(i);
    }
    cout << sum << endl;
    return 0;
}
```
Combining Thread-Private Values

• The need to combine results from each thread to a single value is known as **reduction**

• Reduction can be done:
  – One at a time as in the previous slide
  – In parallel using a tree
Data Parallel Summary

- Variables that will be read and modified by multiple threads are known as **shared variables**
- Access to shared variables must be **synchronized (atomic)** or else results will be corrupted
- Synchronizing access to **shared variables** **hurts** parallel performance
- **Private variables** are **per-thread** variables and need NOT be synchronized
- By using **private variables** to produce results per thread we can then **reduce** those results to the single, desired answer
What Cannot Be Parallelized

- Tasks that are fairly small
  - Starting a thread has some significant overhead
  - If the array is small enough, one thread will be faster
- Parallelization is much harder when some results (produced later) are dependent on other results (produced earlier)
  - Example:
    Prefix sums \( data[i] = \sum_{k=0}^{i} data[k] \)
    - Much harder to split the array between threads because later values need results from previous values

Prefix Sums Example

```c
const int MAX = 1000000;
int data[MAX];

void do_prefix_sum()
{
    for(int i=1; i < MAX; i++)
    {
        data[i] = data[i] + data[i-1];
    }
}

int main()
{
    /* Init data array */
    do_prefix_sum()
    return 0;
}
```

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>2</td>
<td>25</td>
</tr>
<tr>
<td>8</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>38</td>
</tr>
</tbody>
</table>

Before | After
What Can Be Parallelized

• Many audio, image, and video operations apply independent manipulations to all pixels of an image/video or samples of audio

• A whole market of processors (GPUs=Graphics Processing Units) and software libraries and languages exist
  – These GPUs have thousands of "cores" allowing massive parallelism
Up Next: OpenMP

• We will practice apply parallelism using **OpenMP**

• OpenMP is a library (API) that supports
  – Data parallelization of **for** loops (known number of iterations)
    • Automated thread creation
    • Simple specification of **shared** and **private** variables
      – Automated synchronization if shared variables are specified correctly
    • Performing reductions
  – Task parallelization
    • Simply specify what portions of code and execute in parallel

```c
const int MAX = 1000000;
int data[MAX];
void do_sum()
{
    int sum = 0;
    #pragma omp parallel for
    for(int i=1; i < MAX; i++)
    {
        sum += data[i];
    }
}
int main()
{
    #pragma omp sections
    {
        #pragma omp section
        {
            /* Run in parallel... */
        }
        #pragma omp section
        {
            /* ...with this section */
        }
    }
    return 0;
}
```