Unit 18
Performance & Parallelism

18.2
Review of Software Execution (1)

- Recall:
  1. High level code is compiled to ____________ instructions
  2. When executed, CPU uses PC register to know what instruction to ____________ 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 ________ & must be updated

#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;
}

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

18.3
Review of Software Execution (2)

18.4
Improving Performance

- We want to improve the performance of our computation
- Question: What are we referring to when we say "performance"?
  - ____________
  - ____________
  - ____________
- We will primarily consider ____________ in this discussion
How Do We Measure Speed

- **Fundamental Measurement:** _______
  - Absolute time from _______ to _________
  - 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 _______ of the system and it works great from the perspective of the __________ 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 _______ 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 _____________ to get from point A to point B, then the _______
  - This is known as _________ [units of seconds]
  - Time from the start of an operation until it completes
- If _____________ to get from point A to point B, the _____ looks much better
  - This is known as _________ [jobs/second]
- The **overall** execution time (latency) may best be improved by ______________ throughput and not the latency of individual tasks

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 ____________
  - Pipelining: Improves ____________
Caching

- **Cache (def.)** – "to store away in hiding or for future use"
- Primary idea
  - The ______ you access or use something you expend the ______ amount of time to get it
  - However, store it someplace (i.e. in a cache) you can get it more ______ 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 ______ it in the cache)
- Examples:
  - ___________________
  - ___________________
  - ___________________

Cache Overview

- Remember what register are used for?
  - Only a ______ (32 or 64) so that we can access really quickly
  - Controlled by the ______
- Cache memory is a small-ish, (____bytes to a few _____bytes) "_______" memory usually built onto the processor chip
- Will hold ______ of the latest data & instructions accessed by the processor
- Managed by the ______
  - ____________ to the software

Cache Operation (1)

- When processor wants data or instructions it always ______ in the cache first
  - If it is there, ______ access
  - If not, get it from __________
  - Memory will also supply ______ data since it is likely to be needed soon
  - Why?
  - Things like ______ & ______ (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 _______ of larger levels.

Main Memory ~ 100 ns
L2 Cache ~ 10 ns
L1 Cache ~ 1 ns
Registers ~ 1 ns

Unit of Transfer: 8-64 bytes
Faster
Less Expensive
Larger
More Expensive
Slower

Pipelining

- We'll now look at a hardware technique called **pipelining** to improve __________
- The key idea is to __________ 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: _______ ns
  - Can we improve?

\[
\text{Clock Freq.} = \frac{1}{\text{ns}} = \text{MHz}
\]

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).

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

Stage 1

- Stage 1
- Stage 2

<table>
<thead>
<tr>
<th>Stage</th>
<th>Clock Cycle 0</th>
<th>Clock Cycle 1</th>
<th>Clock Cycle 2</th>
</tr>
</thead>
</table>

**Time for 0th elements to complete:**
**Time between each of the remaining 99 element completing:**
**Total:**
**Define:**

\[
speedup = \frac{\text{original time}}{1000\text{ns}} = \frac{1000\text{ns}}{1000\text{ns}} = \text{MHz}
\]

\[
\text{Clock freq.:} = \text{MHz}
\]
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? ____________
• Questions/Issues?
  – ___________ stage delays
  – ___________ of registers (Not free to split stages)
  • This limits how much we can split our logic

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

Non-Pipelined Processors

• Currently we know our processors execute software 1 instruction at a time
• 3 steps/stages of work for each instruction are:
  – __________
  – __________
  – __________

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 ____________
• Overall throughput, and thus total latency, are greatly improved

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

Time for 0th elements to complete: __________
Time between each of the remaining 99 element completing: __________
Total: __________

speedup = 1000ns / 257.5ns = 4x
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 ______; frequency goes _____
- All kinds of interesting issues come up though when we overlap instructions and our discussed in future CENG courses

Clock freq. = 1/10ns = 100MHz

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)

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
  - Power ∝ _______^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 ______ billion operations/second but uses ____ Watts
- Too much power means too much __________
  - 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 ________ operations/second (1 billion each)
  - But now we only require ____ Watts
- To continue improving performance at reasonable ______________ the trend is to parallel processing (multiple processor cores per chip)

Improving Performance

- Processors can only do ______ 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?
    - ________________
- 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

Multi-Core Processors

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

Multithreaded Programs

- OR rather than executing different programs each core can be working collaboratively on the ________ problem
  - This can help us improve the ________ of single program
- This requires us to break our work into tasks (aka_______)
Kinds of Parallelism

- To improve latency we want to start thinking how we can decompose our single program into parallel subtasks
- __________ 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)
- __________ parallelism
  - Perform __________ operations on many independent data elements (Same code operations running on different portions of data)
  - Usually taking iterations of a '__' loop and splitting them up between tasks
  - We will focus on data parallelism in this class

Ideal Parallel Performance Gain

- Define:
  Parallel speedup = ________ 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 ________ times faster
- Ideal parallel speedup with N processor cores is ______
  - Known as ________ speedup
  - It is often hard to achieve ________ speedup
  - Why?
    - __________ 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? ______

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?
  - ____________________________
Shared Variables

- Recall all variables live in memory
- To update a variable (e.g., \( \text{sum} = \text{sum} + \text{data}[i] \)) the CPU must:
  - _______ the variable from memory into a register
  - _______ the value (perform the addition)
  - _______ (save) the new value back to memory

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

```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++)
    {
        sum += data[i]; // RMW cycle
    }
}

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

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 _______ value of sum and each add their respective data element to that ______ 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;
}
```

Synchronized (Atomic) Access

- For variables that are ____________________ by multiple threads we need to _______________ the access
- Each thread should do its full Read-Modify-Write cycle uninterrupted (________) 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]); // Ensure atomicity
    }
}

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 _______ most of the benefit of parallelism
  - To ensure correctness we must update sum one transaction at a time
  - We say that access has been ___________ (1 at a time = same as 1 thread = BAD!)

```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]); // Ensure atomicity
    }
}

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 ___________, variables declared inside the thread are ____________?
  - Private = Each thread has their own ____________ (Thread 0 uses i=0...499,999 while Thread 1 uses i=500,000 to 999,999)
  - Private variables can be accessed in parallel (by bringing them into a ________ or ________ memory local to each core)

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
  - Use a private lsum ("local sum") to sum all the elements a thread is ____________
  - Access to lsum need _________ atomic since each thread has their own
- We need only ________ T (# of threads) local sums as threads ____________

const int MAX = 1000000;
const int T = /* # of threads */;
int data[MAX];
int do_sum(int s, int n)
{
  int lsum = 0;
  for(int i=s; i < s+n; i++){
    lsum += data[i];
  }
  return lsum;
}

int main()
{
  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 can be done:
  - One at a time as in the previous slide
  - In parallel using a ________

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 __________
  - Starting a thread has some significant overhead
  - If the array is small enough, one thread will be __________
- Parallelization is much harder when some results (produced later) are _______________ on other results (produced earlier)
  - Example: Prefix sums $data[i] = _______________
  - Much harder to split the array between threads because later values need results from ____________ values

Prefix Sums Example

```
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];
  }
}
```

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 (_____s=Graphics Processing Units) and software libraries and languages exist
  - These GPUs have thousands of "cores" allowing massive parallelism

Prefix Sums Example

```
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];
  }
}
```

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

```
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];
  }
}
```

```
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];
  }
}
```

```
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];
  }
}
```

```
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];
  }
}
```

```
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];
  }
}
```

```
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];
  }
}
```

```
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];
  }
}
```