CSCI-GA.3033-017

Special Topics:

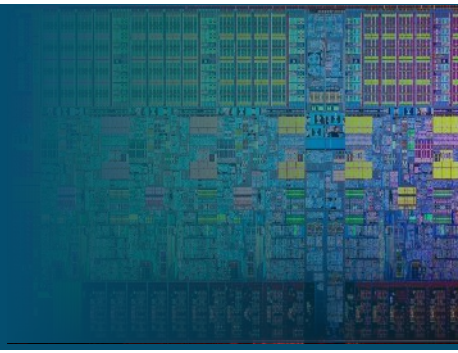
Multicore Programming

Final Exam Review

Christopher Mitchell, Ph.D.

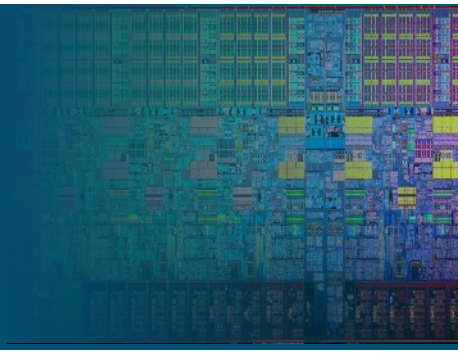
cmitchell@cs.nyu.edu || http://z80.me

Outline



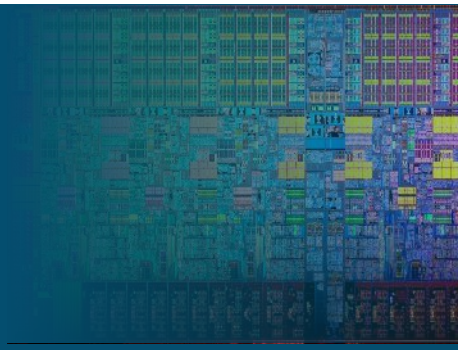
- Topic Review
  - Lectures
  - Labs
  - Midterm Quiz

Lecture Review



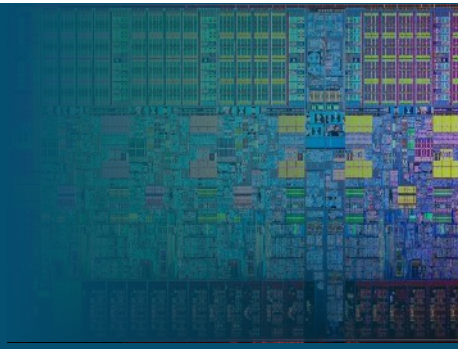
1. The Multicore Revolution
2. Parallelism, Concurrency, and Performance
3. Understanding Hardware
4. Parallel Programming & Pthreads: An Introduction
5. Threads and Pthreads II
6. Coordinating Resources
7. Synchronized Structures I
8. Synchronized Structures II
9. Multicore Correctness
10. Multicore Performance Evaluation
11. Heterogeneous Multicore
12. Transactional Memory

Lecture 1:
The Multicore Revolution



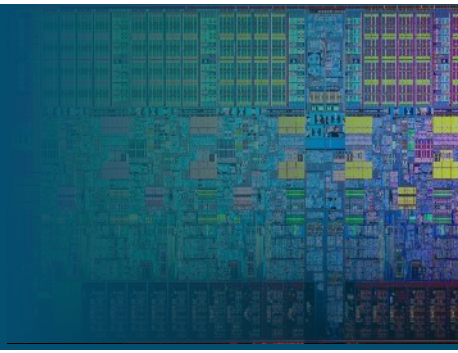
- Do know:
  - The history of CPUs, especially trends in transistors, area, power usage, number of cores, and capabilities
    - Why we stopped making CPUs faster and started making them more parallel instead
  - What an assembly program and an assembly instruction is
  - How pipelining works and what it's for
    - Pipeline acceleration features, pipeline hazards
  - Why we need parallel programming
- Don't:
  - Try to memorize exact transistor counts, names, years, and other minutae that you could just look up in a book. This isn't a test of your ability to memorize.

Lecture 2: Parallelism, Concurrency, and Performance



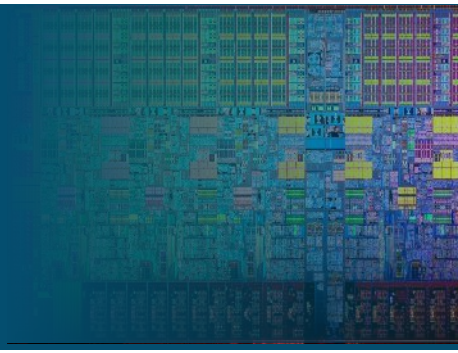
- The difference between parallelism and concurrency
- What Amdahl's law is, means, and how to apply it
- How to model a program as a DAG
  - How to compute work, span, and parallelism
- Types of parallelism
  - Review the homework solutions; this question tripped up a lot of you on the homework
- Flynn classifications, what they mean, and what SIMD, MIMD, SISD, and MISD.
- Parallel programming models and their properties, including (specifically) PRAM, LogP, and BSP

Lecture 3:
Know Your Hardware



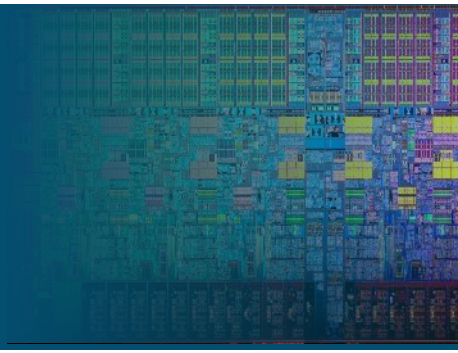
- Do Know:
  - What the memory wall is, what a von Neumann architecture is, and what the bottleneck in a von Neumann architecture is.
  - What spatial and temporal locality are
  - Why we need CPU cache, how it works, and its history in brief
  - Consistency and coherence in CPU caches, including the protocols we discussed
- Don't memorize:
  - Number of AGUs in Intel Haswell architecture
  - The exact MESI coherence protocol (but **do** understand why each transition make sense and where each comes from!)

Lecture 4: Parallel Programming and Pthreads



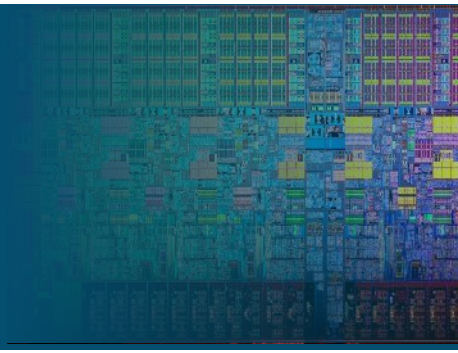
- How a program model needs to define running tasks
- What a coroutine is and how it works
- The basics of the fork/join paradigm
- Types of synchronization and communication: message passing and shared memory
- Threads: What they are (and are not)
  - The stack, the heap, and threads
  - What data races, critical sections, and locks are
- What a mutex is and how to use it
- What a condition variable is and how to use it
- What a semaphore is and how to use it
- How we built a simple concurrent queue with locks

Lecture 5:
Threads and Pthreads II



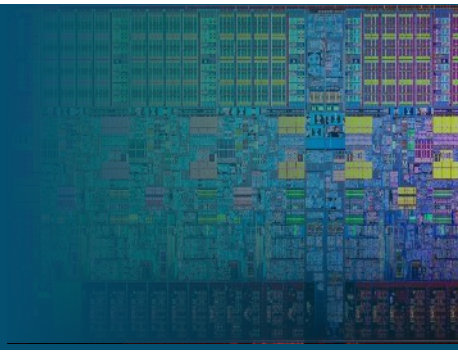
- Pthread demo: how to create and launch pthreads
- How to create a mutex given only atomic variable accesses via Peterson's Algorithm
  - Limitations of Peterson's Algorithm
- How semaphores work
  - How to make a mutex with semaphores
- How condition variables work
  - How to make a condition variable with semaphores
- What a lost wakeup condition is, and how to fix it

Lecture 6: Coordinating Resources



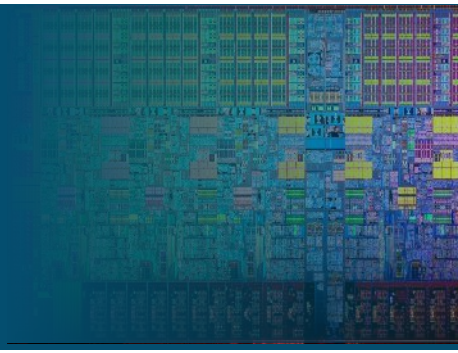
- Do know:
  - What reader-writer locks are, how they help performance, and how to use them
  - Review our evolution of creating a reader-writer lock: the naïve design, solving writer starvation, then solving reader starvation
  - How pthread reader-writer locks are used
  - What barriers are
  - How our implementation of barriers with semaphores works
- Don't memorize
  - The exact code for our reader-writer lock implementation or our barrier implementation

Lecture 7: Coordinating Resources, Part I



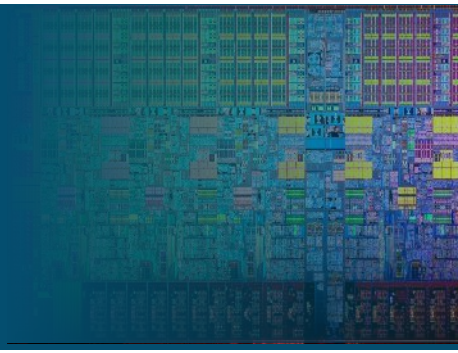
- How to build a concurrent queue
  - What linearizability is and how it helps
    - How a sentinel gives a locking or lock-free queue linearizability
  - How simultaneous enqueueing and dequeing works in a locked, sentinel-based concurrent queue
  - Making the queue lock-free
    - What CAS is, and how to use it
    - How we maintain linearizability

Lecture 8: Coordinating Resources, Part II



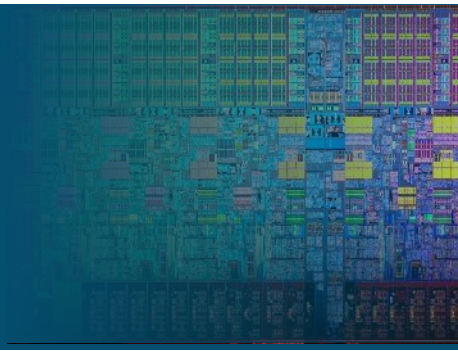
- What the ABA problem is, and how to solve it
- What `volatile` means in C++.
- How a concurrent ordered list can be built
  - Locking version: how to use lock coupling
  - How to make insertion and deletion lock-free
- How a concurrent hash table can be built
  - How ordered lists can be turned into a hash table
  - What a split ordered list is
  - How split ordered lists make concurrent resizing easy

Lecture 9: Multicore Correctness



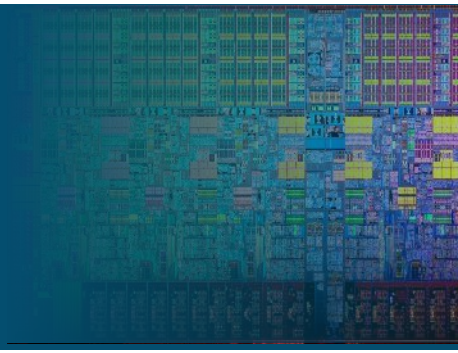
- Know about the different types of real-world multithreaded programming bugs, and what each one is.
  - Remember that many bugs can fit into multiple categories!
- Software approaches to detecting concurrency bugs
  - How each approach works, pros and cons
  - What a lockset is, and how it helps
- Preventing concurrency bugs: understand the argument for deterministic execution (and why it's hard in practice)

Lecture 10: Multicore Performance Evaluation



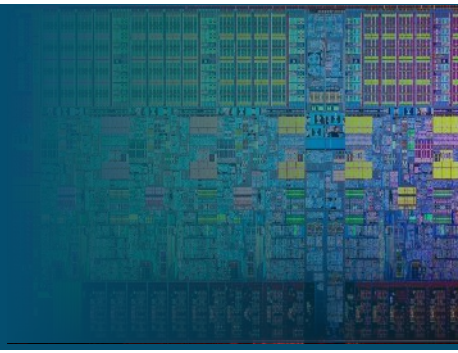
- Do:
  - Understand why it might be important to measure latency and/or throughput
    - How do more cores or faster cores affect each one?
  - Understand the relationship between execution time, clocks per instruction, instruction count, and cycle time
  - Know what an ISA is, and how different CPUs with the same ISA or different ISAs relate to each other
  - Understand why it may be hard to get a single value for how long a multicore program takes to run
  - Understand ways to measure multicore program throughput and turnaround time
  - Know why benchmarks are important
  - Understand why multicore program bottlenecks can be time-dependent
- Do not:
  - Memorize the notation for multicore program throughput and turnaround time: understand *why* that notation is used instead
  - Memorize the applications in each benchmark

Lecture 11: Heterogeneous Multicore



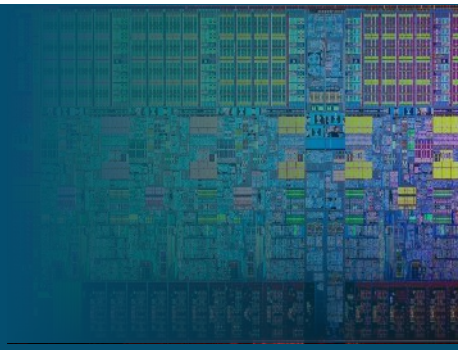
- Understand how to classify heterogeneous multicore CPUs based on the number of core types and variation in ISA types
- Understand how heterogenous multicore CPUs can complicate scheduling
- Remember properties of a few interesting real-world and research heterogeneous multicore CPUs
  - Understand how the Cell CPU schedules jobs on SPEs
  - Understand FUSE and SPLIT in the Core Fusion
  - Understand caching and communication challenges
- Understand how you might consider the problem of load balancing (scheduling) for heterogeneous multicores, and how this applies to homogeneous multicores
  - Thread costs, thread dependencies, and locality

Lecture 12: Transactional Memory



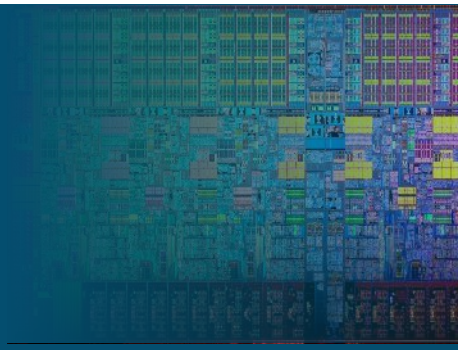
- Understand the properties of transactions (for databases and transactional memory)
  - What ACID is and what it provides
- How transactions can be used in C++
  - The difference between atomic and relaxed transactions
- What optimistic and pessimistic concurrency control are, and the properties of each
  - How to detect and handle conflicts
- How software transactional memory works
- How hardware transactional memory works
- The pros and cons of HTM and STM

Lab Review



- Pro tip: review your write-ups and lab notes, not the lab assignments. Sample questions to ask yourself:
- Lab 1: What were the limitations of using a lock to maintain correctness in a simple concurrent queue?
- Lab 2: How did you implement a thread worker pool? How else could you have done it? How did sockets and connections complicate sending jobs to workers? How did you maximize performance?
- Lab 3: How were you able to measure performance? How does performance measurement affect the measured performance (and how can you minimize this)? What are the relative merits of reporting median, average, minimum, and maximum latency? Throughput?
- Lab 4: How did you maintain performance in the face of slow disk accesses? How did you (and how else could you) guarantee consistency between an in-memory cache and an on-disk store?

Midterm Quiz Review: Q1



Thread safety. Consider the following presumably thread-unsafe Bitmap class and its set(), reset(), and test() methods. It stores an array of Booleans, and either flips (inverts) or resets (sets to false) each element of the underlying array. Assume that the compiler will be creating an array of *bytes*, one byte per element, even though that wastes 7/8ths of the bits. Answer each of the following 5 sub-parts separately.

- Re-write flip() and reset() to be thread-safe *as simply as possible*. Mention any additional private member variables that need to be added to the class, and/or any additional initialization necessary in Bitmap::Bitmap().
- Using the same *simple* technique to make test() thread-safe.
- Propose one or two ways to make this class support more parallelism (text is required; code or pseudocode is optional). What access pattern(s) do your techniques assume, if any?
- If I later tell you that we're only going to be targeting x86 or x86-64, would that simplify (a) – (c)? Why? What would you have done instead?
- What if instead of using one byte per array element, the compiler decided to be clever and packed 8 elements into each byte (i.e., 1 bit per element)? Again, code is not necessary here, only a thoughtful description.

```
class Bitmap {

  public:
    BitMap(const size_t capacity) {
        store_ = new bool[capacity];
    }
    ~BitMap(void) {
        delete[] store_;
    }
    void flip(const size_t idx) {
        store_[idx] = !store_[idx];
    }
    void reset(const size_t idx) {
        store_[idx] = false;
    }
    const bool test(const size_t idx) {
        return store_[idx];
    }

  private:
    bool *store_;
};
```

Midterm Quiz Review: Q2

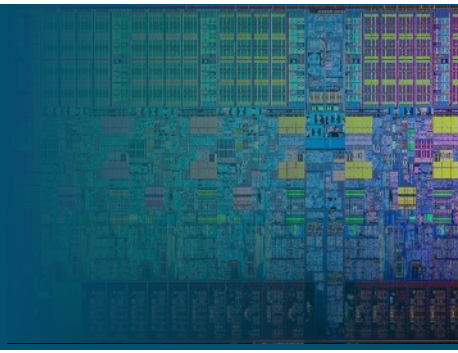
Compare and Swap and Lock-Free Thread Safety.

- What does Compare and Swap do? Fill in:
  __*(return val)*__ CAS(__*(arguments)*__)
  and explain what it does.
- Consider the following function simplified from something you saw in your homework, that aggregates a *single* statistic. Is it thread-safe as-is? If yes, why? If not, how can you make it thread-safe using CAS?

```
1     static double sum_stat_a = 0;
2     int aggregateStats(double stat_a) {
3           sum_stat_a += stat_a;

4           return sum_stat_a;
5     }
6     void init(void) { }
```

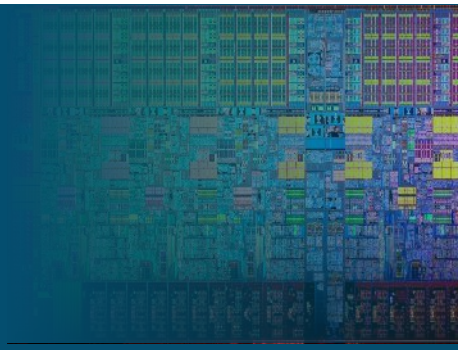
Midterm Quiz Review: Q3



Parallelism. Describe the following types of parallelism in a sentence or two, and mention whether they're generally introduced into a program by the CPU, the compiler, and/or the application programmer:

- Task-level parallelism
- Basic block-level parallelism
- Instruction-level parallelism

Midterm Quiz Review: Q4



Coroutines. Describe coroutines, what they're used for, and how they differ from threads. If you'd like, sketch out the pseudocode for a simple coroutine and how it would run (optional). If you can't remember what a coroutine is, for a maximum of half the available points for this question, explain the ABA problem instead.

Bonus: What is hyperthreading?