Speed-Up

Peter Chapin

CIS-4230, Parallel Programming

Vermont State University

“What Happened To My Speed-Up?”

- There are several reasons why speed-up might be poor…
  - “Hyperthreading” isn’t helping (much)
  - Amdahl’s Law
  - Overhead of…
    - … thread creation/destruction
    - … thread synchronization
  - Thread interference
    - Threads are blocked too much waiting for each other. This is different than overhead which is about the time spent actually changing a thread’s state. Overheads are generally small, but a poorly designed program can have large wait times.
  - **Memory access issues** ← *Usually the biggest problem*

Hyperthreading

- "Hyperthreading" is an Intel marketing term
  - … but other process vendors have a similar technology
- Create two (or more) register files in the CPU
  - … this allows two (or more) threads to execute "simultaneously."
- Let the threads share the functional units (FUs)
  - Functional Unit: ALU, address calculator, FPU, barrel shifter, etc.
  - Typically, multiple function units of the same type (e.g., two ALUs)
  - The program does not use all the FUs at the same time
  - The second thread can use the FUs that would otherwise be idle

Hyperthreading Doesn’t Work (very well)

- Problems:
  - A well-coded program (at the assembly language level) orders instructions so that their overlapping execution keeps as many FUs as busy as possible
    - … so there really aren’t that many “idle” FUs.
    - Typically, compilers write assembly language, so this depends on good compilers
    - It also depends on the sort of program one is trying to run.
  - As a result, hyperthreads are often stalled, waiting for an available FU
  - Intel states speed-ups of 1.3 might be typical in a “good situation.”
    - In practice speed ups are even less… and might hover around 1.1.
- Why bother?
  - It is easy to implement, so why not? Plus, it sounds good on paper.

Hyperthreading Works When…

- Conditions favorable to hyperthreading:
  - Lousy compilers that don’t order instructions well
    - … but the CPU itself might be able to compensate for that due to “out of order” execution features.
    - … so a cheaper CPU might also be necessary for effective hyperthreading (ironically).
  - Threads that do very different things
    - e.g., one thread doing intensive floating point (and thus using the FPU) while another thread does strictly integer calculations on the ALU(s) and address calculator(s), etc.
    - *On a typical system, this might actually come up. It is less likely in a parallel programming context.*

Lemuria

- Consider Lemuria…
    - 2 processors x 4 cores/processor x 2 hyperthreads/core = 16 threads
    - Speed-up of 16?
        - Not likely!
        - Probably closer to 8-10, even under ideal conditions (i.e., no other issues causing speed-up problems)
        - … especially when all threads try to use the FPUs… lots of stalling of the hyperthreads waiting for access to the floating point unit(s).
    - In fact, the system overhead of managing the extra “nearly useless” threads might be greater than whatever small benefit they provide
        - Try forcing the thread count to 8 instead of using the system-reported count of 16. *Performance might actually be better!*

Amdahl’s Law

- Fundamental Issue:
  - Programs typically have a serial portion and a parallelizable portion. Even if the parallelizable portion is made to execute “instantly” (lots of threads), the serial portion runs at the same speed as before.
  - Thus… the maximum speed-up is bounded.
  - Example:
    - Serial portion: 25% of execution time
    - Parallelizable portion: 75% of execution time (when run in a single thread)
    - Maximum speed-up = 4 no matter how many processors are applied.

Example Continued

- The good news is that typically (hopefully!), the parallelizable portion grows more rapidly as the problem grows. Thus…
  - Serial portion: O(n)
  - Parallelizable portion: O(n^2)
  - Now double the problem size…
    - Serial portion: 2*25% of original execution time
    - Parallelizable portion: 4*75% of original execution time
    - If the Parallelizable portion executes “instantly” due to aggressive parallelization, the speed-up becomes: ((2 * 0.25) + (4 * 0.75)) / (2 * 0.25) = **7**
  - We tend to only care about large problems. Thus, Amdahl’s Law isn’t scary.
    - *Provided the asymptotic growth of the parallelizable portion is large.*

Thread Overhead

- It takes time to manipulate threads
  - The system is involved when creating and destroying threads
    - They are hardware entities. User mode threads offer no real parallelism
  - The system is involved when threads synchronize
    - Mutex objects, condition variables, barriers, etc.
    - Suspending a thread and then finding and starting another one is significantly complex.
  - If the work done between synchronization operations is too small…
    - The time spent managing the threads will be a large percentage, and speed-up suffers
- BUT…
  - Usually, this overhead shrinks (as a percentage) as the problem size grows.

Thread Interference

- If a thread stalls waiting for something to happen…
  - … a processor is underutilized. Speed-up suffers
- Consider barriers:
  - A team of threads executes a for loop in parallel, with each thread doing a subset of all the loop's iterations.
  - Suppose one thread finishes early
    - Its work unit is easier than the others for some reason
  - That thread waits at the barrier, <u>doing nothing</u> until the other threads finish.

Memory Issues

- The BIG ONE!
  - In the old days, processor performance was what held things back
  - Today it is the limited memory bus bandwidth.
- The memory hierarchy
  - Registers: access time 1 ns or less
  - L1 cache: access time ~10ns
  - L2 cache: access time ~25ns
  - Main memory: access time ~100s of ns
- Reading a value from main memory can take, literally, 100 times as long as reading a value from a register!

Caching

- Without the caches, the CPUs we have would be pathetically slow
- But… caches depend on “locality of reference” to work.
    - Values are reused often
    - Values close to each other in memory are often used together
    - Values are reused closely in time
- The frequently used values are stored in a (small, fast(er)) cache where they can be accessed more quickly.
    - *This assumes the cache can hold all such values!*

Terminology

- Cache “hit”: when the cache satisfies a memory access.
    - This is still likely 5-10x slower than accessing a register.
- Cache “miss”: when the desired value is not in the cache
    - Value fetched from main memory (or a lower level cache)… very… very… slowly… and then stored in the cache for later.
    - Processor stalled while waiting for the memory access
        - … or maybe not. It might be able to execute other upcoming instructions while it waits
- Cache “line”: Values are fetched from main memory in “lines.”
    - … might be 8 or 16 bytes at a time… or more. Depends on the cache architecture. Thus, a miss might pre-fetch values we will need soon

Complex

- Cache Design is a highly complicated topic
- A topic for a computer architecture course
- Many designs exist with various properties

Hugely influences performance!

Especially when multiple threads are competing for access to the same memory

Multi-Core CPUs

- Various cache options
  - Each core has its own L1 and L2 caches.
    - Accessing data in cache does not influence the other core
    - … but if a core brings a value into its cache, it doesn’t help the other core either
    - Flushing a value out of the cache does not affect values in the other core’s cache
  - Each core has its own L1 cache but the cores share an L2 cache
    - Access to L2 might stall the other core
    - … but if a core brings a value into L2, the other core can get it without main memory
    - Flushing a value out of the cache prevents the other core from getting it (at least not without main memory access)

Multiple CPUs

- Rather different situation…
  - No cache sharing
    - No fetching a value the other CPU can use
    - No flushing values the other CPU needs
  - CPUs can still stall each other from accessing main memory
- Lemuria has multiple CPUs that are each multi-core
  - Creates a very complex caching environment
  - Hard to predict, analyze, and explain behaviors.

Cache Coherency

- Suppose two CPUs store the same value in their independent caches
  - Now, suppose one of the CPUs modifies that value
  - How does the other CPU know to read the modified value? It still has the original value in its cache!
- Cache Coherency hardware deals with this
  - Many designs (refer to a computer architecture textbook)
  - Basically, the caches must communicate to ensure only the most recent value is used. This creates hardware complexity and/or overheads.
  - Further complicates the analysis

Where Does This Leave Us?

- Confused!
  - The complexity of caching makes understanding behaviors difficult
- Tools Can Help
  - Intel Parallel Studio (https://software.intel.com/en-us/parallel-studio-xe)
  - Eclipse Parallel Tools Platform (https://www.eclipse.org/ptp/)
- Experimentation Helps
  - But… regardless of the approach, a perfectly optimized program will likely only be perfectly optimized for one particular CPU/Cache architecture. Even a different model of the same CPU family will likely behave differently.