Lecture 3
Know Your Hardware

Christopher Mitchell, Ph.D.
cmitCHELL@cs.nyu.edu || http://z80.me
Lecture 3 Outline

➢ Parallel Programming Model Recap
➢ Coprocessors
➢ Multicore Hardware
➢ Performance and Hardware
➢ Diet Threads
Parallel Programming Models Recap

• Express implementation of software for hardware
  • Portability: software made for the hardware, then hardware made for the software
  • “Bridges” the two

• Models have implicit generality and performance

• Classification
  • Process interaction (shared memory, message passing)
  • Problem decomposition (task, data)

<table>
<thead>
<tr>
<th>Name</th>
<th>Interaction</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRAM</td>
<td>Shared Memory</td>
<td>Data</td>
</tr>
<tr>
<td>LogP</td>
<td>Message Passing</td>
<td>(Unspecified)</td>
</tr>
<tr>
<td>BSP</td>
<td>(Ambiguous)</td>
<td>Task</td>
</tr>
</tbody>
</table>
Coprocessors

• Optional chip supplementing CPU
• Catches specific instructions in the instruction stream
• Historical coprocessors
  • FPU
    • 8087: FPU coprocessor for 8086/8088
  • Video
  • I/O
    • 8089
• Modern coprocessors
  • Audio codecs
  • GPUs!
  • Physics
  • ANN
Multicore Hardware
Computer Technology

• Memory
  • DRAM capacity: 2x / 2 years (since ‘96)
  • 64x size improvement in last decade.

• Processor
  • Speed 2x / 1.5 years (since ‘85)
  • 100x performance in last decade

• Traditional Disk Drive
  • Capacity: 2x / 1 year (since ‘97)
  • 250x size in last decade
Memory Wall

“Moore’s Law”

Processor-Memory Performance Gap:
(grows 50% / year)

Most of the single core performance loss is on the memory system!
von Neumann Bottleneck

Processor $ Memory

Cache
Two Main Data Characteristics

• **Temporal Locality**
  • I used X
  • Most probably I will use it again soon

• **Spatial Locality**
  • I used item number M
  • Most probably I will need item M+1 soon
Cache Analogy: I’m Hungry!

• Option 1: Go to refrigerator (L1 Cache)
  • Found → eat!
  • Latency = 1 minute

• Option 2: Go to store (L2 Cache)
  • Found → purchase, take home, eat!
  • Latency = 20-30 minutes

• Option 3: Grow food! (Main Memory)
  • Plant, wait ... wait ... wait ... , harvest, eat!
  • Latency = ~250,000 minutes (~ 6 months)
Storage Hierarchy Technology

Source: Ryan J. Leng
Why Memory Wall?

• DRAMs not optimized for speed but for density (till now at least!)
• Off-chip bandwidth
• Increasing number of on-chip cores
  • Need to be fed with instructions and data
  • Big pressure on buses, memory ports, ...
Cache Memory: Yesterday

• Processor-Memory gap not very wide
• Simple cache (one or two levels)
• Inclusive
• Small size and associativity
Cache Memory: Today

- Wider Processor-Memory gap
- Two or three levels of cache hierarchy
- Larger size and associativity
- Inclusion property revisited
- Coherency
- Many optimizations
  - Dealing with static power
  - Dealing with soft-errors
  - Prefetching
  - ...
Cache Memory: Tomorrow

- Very wide processor-memory gap
- Multiple cache hierarchies (multi-core)
- On/Off chip bandwidths become bottleneck
- Scalability problem
- Technological constraints
  - Power
  - Variability
  - ...
100s On-Chip Cores

• Technologically possible

• Near-future usage:
  • Massively parallel applications
    • Multithreading

• In the long run
  • Day to day use
    • Hybrid multithreading + multiprogramming
From Single Core to Multicore

• Currently mostly shared memory
  • This can change in the future
  • The “sharing” can be logical only (i.e. distributed shared memory)

• A new set of complications, in addition to what we already have
  • Coherence (all cores see same data; keep reading)
  • Consistency (policy for ordering of memory accesses)
Shared Memory Multicore

- Uniform
  - Uniform Cache Access
  - Uniform Memory Access
- Non-Uniform
  - Non-Uniform Cache Access
  - Non-Uniform Memory Access
Memory Model

- **Intuitive**: Reading from an address returns the most recent write to that address.
- This is what we find in uniprocessors
- For multicore, we call this: *sequential consistency*
  - Much harder and tricky to achieve
  - This is why we need *coherence*
Sequential Consistency Model

• Example:
  • P1 writes data=1, then writes flag=1
  • P2 waits until flag=1, then reads data

<table>
<thead>
<tr>
<th>If P2 reads flag</th>
<th>Then P2 may read data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Ensuring Consistency: Coherence Protocol

• Cache coherence needed in multicore processors to ensure consistency

• A memory system is coherent if:
  • P writes to X; no other processor writes to X; P reads X and receives the value previously written by P
  • P1 writes to X; no other processor writes to X; sufficient time elapses; P2 reads X and receives value written by P1
  • Two writes to the same location by two processors are seen in the same order by all processors – write serialization
Cache Coherence

c0, d0, y0: Privately owned by Core 0
a1, b1, y1, z1: Privately owned by Core 1
Initial condition: x = 2;  // (shared variable)

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y0 = x;</td>
<td>y1 = 2 * x;</td>
</tr>
<tr>
<td>1</td>
<td>x = 5;</td>
<td>a1 = b1;</td>
</tr>
<tr>
<td>2</td>
<td>c0 = d0;</td>
<td>z1 = 3 * x;</td>
</tr>
</tbody>
</table>

y0 = 2
y1 = 4
z1 = ???
Snooping Cache Coherence

• The cores share a bus

• Any signal transmitted on the bus can be “seen” by all cores connected to the bus.

• When core 0 updates the copy of $x$ stored in its cache it also broadcasts this information across the bus.

• If core 1 is “snooping” the bus, it will see that $x$ has been updated and it can mark its copy of $x$ as invalid.
Directory Based Cache Coherence

• Uses a data structure called a directory that stores the status of each cache line.

• When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated.
Cache Coherence Protocols

- Snooping protocols
  - Write invalidate
- Directory-based protocols
  - Write update
Example: MESI Protocol

PR = processor read  
PW = processor write  
BR = observed bus read  
BW = observed bus write  
S/~/S = shared/NOT shared
The Future In Technology

- Traditional
  - SRAM
  - DRAM
  - Hard drives

- New
  - eDRAM
  - Flash
  - Solid-State Drive

- Even Newer
  - (disruptive technology?)
    - M-RAM
    - STT-RAM
    - PCM
    - ...
As A Programmer

• A parallel programmer is also a performance programmer: know your hardware.

• Your program does not execute in a vacuum.

• In theory, compilers understand memory hierarchy and can optimize your program;
  • In practice they don’t!!

• Even if compiler optimizes one algorithm, it won’t know about a different algorithm that might be a much better match to the processor
As A Programmer

• You don’t see the cache
  • But you feel it

• You see the disk and memory
  • So you can explicitly manage them
As A Programmer: Tools In Your Box

- Tiling
- Number of threads you spawn at any given time
- Thread granularity
- User thread scheduling
- Locality (both types)
- What is your performance metric?
  - Throughput
  - Latency
  - Bandwidth-delay product
- Best performance for a specific configuration vs. scalability
Performance and Hardware
Eg: Intel Haswell (2014)

- New microarchitecture
- 22nm (Broadwell: 14nm)
Eg: Intel Haswell (2014)

- New microarchitecture
- 22nm (Broadwell: 14nm)
Intel Haswell Architecture

• Improvement over its predecessor Ivy Bridge
  • 3->4 ALUs (Arithmetic Logic Units)
  • 2->3 AGUs (Address Generation Units)
  • 1->2 Branch Execution Units
  • Partitioned -> Shared instruction decode cache
  • [Disabled] Hardware Transactional Memory support

• Targeting multimedia applications
  • Introduced Advanced Vector Extensions 2 (AVX2)
Features for You to Use

- Sandy bridge processors have 256bit wide vector units per core
- As a programmer you can:
  - Using AVX instructions
  - Use the compiler to vectorize your code
    - http://ispc.github.com/
Two Challenges

Power

Performance

Efficiency

Locality

Data movement costs more than computation.
Your Parallel Program

- **Threads**
  - Granularity
  - How many?

- **Thread types**
  - Processing bound
  - Memory/I/O bound

- **What to run? When?**
  - Where?

- **Communication**

- **Degree of interaction**
What to Do About Caching and Prefetching?

• Use arrays as much as possible. Lists, trees, and graphs have complex traversals which can confuse the prefetcher.

• Avoid long strides. Prefetchers detect strides only in a certain range because detecting longer strides requires a lot more hardware storage.

• If you must use a linked data structure, pre-allocate contiguous memory blocks for its elements and serve future insertions from this pool.

• Can you re-use nodes from your linked-list?
Thought-Provoking Questions

• Can you design your program with different type of parallelism?

• Your code does not execute alone. Can you do something about it to avoid interference?

• As a programmer, what can you do about power?
Conclusions

• More details about the big picture help
  • Number of cores and SMT capability
    • Dynamic adaptation
  • Interconnection
  • Memory hierarchy
  • What is available to software and what is not
Conclusions

• Major bottlenecks
  • Memory
  • Interconnect

• Actual performance of program can be a complicated function of the architecture
  • Slight changes in the architecture or program change the performance significantly

• The art of delegation
  • What to do at user level and what to leave for the compiler, OS, and runtime
What is Threading?

**Process**
- Unique private address space
- Execution stack
- Kernel-level unit of execution
- Communication: Shared memory or pipes

**Thread**
- Shared address space
  - Thread-local stack
- Execution stack
- Kernel- or user-level unit of execution
- Communication: shared memory, pipes, mutexes, condition variables, semaphores
Thread Concepts: Threads

- In this course: Pthreads
  - Shared memory, shared file pointers: need explicit synchronization
- Create
- Join or detach
**Thread Concepts: Mutices**

<table>
<thead>
<tr>
<th>Threads</th>
<th>Mutices</th>
<th>Condition Variables</th>
<th>Semaphores</th>
</tr>
</thead>
</table>

- **Mutex:** “Mutual Exclusion”
- Lock that can be used for **exclusive** access to any shared resource(s)
- Programmer defines what is protected
  - Programmer responsible for locking/unlocking around access to protected resource
Thread Concepts: Mutices

- Using a mutex
  - Lock mutex (waits until lock is released)
  - Use shared resources
  - Unlock mutex
  - *Do not* use shared resource until lock is re-acquired
Thread Concepts: Condition Variables

- Synchronization based on data values
- Always used with mutex
- Replaces:
  1. Lock mutex
  2. Check value of data
  3. Unlock mutex, repeat.
Thread Concepts: Semaphores

- Counting mutex
- Atomically increase or decrease
- Sample use: communication management
  - `sem_post()` (atomically increment) when sending new message to receiver
  - `sem_wait()` (atomically decrement) to receive (or wait for) new message from sender
Diet Threads Conclusion

• Threads: multiple “processes” with tighter integration in single process
• Simultaneous power and danger of shared resources
• Primitives to manage shared access and communication