Outline

- MP Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Consistency, Coherency, Write Serialization
- Write Invalidate Protocol
- Example
- Conclusion

Uniprocessor Performance (SPECint)


- VAX: 25%/year 1978 to mid-1980 (architectural and organizational idea)
- RISC + x86: 52%/year 1986 to 2002; 22%/year 2002 to present

Growth in Clock Rate of Microprocessors
Déjà vu all over again?

- “... today's processors ... are nearing an impasse as technologies approach the speed of light.” - David Mitchell, The Transputer: The Time Is Now (1989)
- Transputer was premature
- Custom multiprocessors strove to lead uniprocessors.
- 2X perf / 1.2 yr
- “We are dedicating all of our future product development to multi-core designs. ... This is a sea change in computing” - Paul Otellini, President, Intel (2004)
- Difference is all microprocessor companies switch to multi-core (AMD, Intel, IBM; all new Apple 2-4 CPUs)
- 2X sequential perf. / 5 yrs
- Biggest programming challenge: 1 to 2 CPUs

<table>
<thead>
<tr>
<th>Manufacturer/Year</th>
<th>AMD/'05</th>
<th>Intel/'06</th>
<th>IBM/'04</th>
<th>Sun/'05</th>
<th>Sun/'07</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/chip</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Threads/chip</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Flynn’s Taxonomy

- Flynn classified by data and control streams in 1966

| Single Instruction Stream, Single Data Stream (SISD) (Uniprocessors) | Single Instruction Stream, Multiple Data Stream (SIMD) (Single ProgCtr: CM-2) |
| Multiplication Instruction Stream, Single Data Stream (MISD) (??? Arguably, no designs) | Multiple Instruction Stream, Multiple Data Stream (MIMD) (Clusters, SMP servers) |

- SIMD ⇒ Data Level Parallelism (problem: locked step)
- MIMD ⇒ Thread Level Parallelism (independent steps)
- MIMD popular because
  - Flexible: N programs or 1 multithreaded program
  - Cost-effective: same MicroProcUnit in desktop PC & MIMD

Other Factors ⇒ Multiprocessors Work Well

- Growth in data-intensive applications
- Databases, file servers, web servers, ... (All: many separate tasks)
- Growing interest in servers, server performance
- Increasing desktop performance less important
- Outside of graphics
- Improved understanding in how to use multiprocessors effectively
- Especially servers, where significant natural TLP (separate tasks)
- Huge cost $$$ advantage of leveraging design investment by replication
  - Rather than unique designs for each higher performance chip
    (a fast new design costs billions of dollars in R&D and factories)

Back to Basics

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Processor Architecture + Communication Architecture
- Two classes of multiprocessors W.R.T. memory:
  1. Centralized Memory Multiprocessor
     - < few dozen processor chips (and < 100 cores) in 2006
     - Small enough to share a single, centralized memory
  2. Physically Distributed-Memory Multiprocessor
     - Larger number of chips and cores than centralized class 1.
     - BW demands ⇒ Memory distributed among processors
     - Distributed shared memory ≤ 256 processors, but easier to code
     - Distributed distinct memories > 1 million processors
     - (Shared address space versus separate address spaces)
Centralized vs. Distributed Shared Memory

Centralized Memory (Dance Hall MP)
(Bad: all memory access delays are big)

Distributed Memory
(Good: most memory accesses are local & fast)

Centralized Memory Multiprocessor
• Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
• Large caches and a single memory can satisfy the memory demands of small number (<17) of processors using a single, shared memory bus
• Can scale to a few dozen processors (<65) using a crossbar (Xbar) switch and many memory banks
• Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases

Distributed Memory Multiprocessor
• Pro: Cost-effective way to scale memory bandwidth
  • If most memory references are to local memory and if much less than 10% of all memory references are writes to shared variables
  • Pro: Reduces latency of local memory accesses
• Con: Communicating data rapidly between processors needs more complex hardware
• Con: Must change software to take advantage of increased memory BW

Models for Communication and Memory Architecture
• Communication occurs by explicitly passing (high latency) messages among the processors: message-passing multiprocessors
  • e.g., warehouse-scale computers: memory of a process cannot be directly accessed by another processor
• Communication occurs through a shared address space (via loads and stores): distributed shared memory multiprocessors either
  • UMA (Uniform Memory Access time) for shared address, centralized memory MP; Symmetric shared-memory architecture
  • NUMA (Non-Uniform Memory Access time multiprocessor) for shared address, distributed memory MP; Distributed shared-memory (DSM) architecture
  • (In past, confusion whether “sharing” meant sharing physical memory UMA {Symmetric MP "dancehall"} or sharing address space NUMA)
Challenges of Parallel Processing

- First challenge is % of program that is inherently sequential
- For 80X speedup from 100 processors, what fraction of original program can be sequential?
  a. 10%  
  b. 5%  
  c. 1%  
  d. 1/4%  

Amdahl’s Law

\[ \text{Speedup} = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}} \]

\[ 80 \times \left( 1 - \text{Fraction}_{\text{parallel}} \right) + \frac{\text{Fraction}_{\text{parallel}}}{100} = 1 \]

\[ 79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}} \]

\[ \text{Fraction}_{\text{parallel}} = \frac{79}{79.2} = 99.75\% \]

Insufficient parallelism!!

Challenges of Parallel Processing

- Challenge two is long latency to remote memory
- Suppose 32 CPU MP, 3.3GHz, 200 ns (= 666 clocks) remote memory, all local memory references are hit in the cache (assume HT = 0), and base CPI is 0.5.
- How much faster if 0.2% of instructions that access remote data were local access?
  a. 1.4X  
  b. 2.0X  
  c. 3.6X  

\[ \text{CPI}_{0.2\%} = \text{Base CPI}_{0} + \text{Remote request rate} \times \text{Remote request cost} \]

\[ \text{CPI}_{0.2\%} = 0.5 + 0.2\% \times 666 = 0.5 + 1.3 = 1.8 \]

No remote communication is 1.8/0.5 or 3.6 times faster than if 0.2% of instructions access one remote datum.

long-latency remote communication!!

Solving Challenges of Parallel Processing

- Application parallelism \( \Rightarrow \) primarily need new algorithms with better parallel performance
- Long remote latency impact \( \Rightarrow \) work for both the architect and the programmer
  - Reduce frequency of remote accesses either by
    - Caching shared data (HW)
    - Restructuring the data layout to make more accesses local (SW)
  - Will discuss first methods to reduce memory access latency via local caches

Symmetric Shared-Memory Architectures

- Evolve from “multiple boards on a shared bus” to multiple processor “cores” on a single chip
- Caches store both
  - Private data used by a single processor
  - Shared data used by multiple processors
- Caching shared data can reduce:
  - Latency to shared data
  - Memory bandwidth for shared data, and
  - Interconnect bandwidth needed
  - But, introduces cache coherence problem
Cache Coherence Problem: P₃ Changes U to 7

- Processors see different values for u after event 3 (new 7 vs. old 5)
- With write-back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - Processes accessing main memory may see very “stale” values
- Unacceptable for programming; writes to shared data often critical!

Example of Memory Consistency Problem

<table>
<thead>
<tr>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Assume initial values of A and flag are 0</em>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A = 1; while (flag == 0); /<em>spin idly</em>/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>flag = 1; print A; /<em>P₂ not cache A until flag = 0</em>/</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Expected result not guaranteed by cache coherence
- Expect memory to respect order between accesses to different locations issued by a given process
- Also, expect memory to preserve orders among accesses to same location by different processes
- Cache coherence is not enough!
  - pertains only to a single location

Intuitive Memory Model

- Reading an address should return the last value written to that address
  - Easy in uniprocessors {except for DMA changes to I/O buffers}

- Too vague and simplistic; two issues
1. Coherence defines values returned by a read
2. Consistency determines when each written value will be returned by a read from the same or a different PU.
- Coherence defines behavior for one location, and
- Consistency defines behavior for location sequences