Concurrent and Distributed Programming

Lecture 2
Parallel architectures

References:
Based on: Mark Silberstein, 236370, Winter 2010
Chapters 2,5 in “Intro to Parallel Computing”, Grama
Lecture 2: CS149 Stanford by Aiken&Oluktun
Last lecture

• Why parallel programming?
• Problem partitioning: SIMD vs. MIMD
• Memory architecture: UMA, NUMA and distributed
• Memory-optimized programming
• Control: task management overhead
• Synchronization
• Granularity and load balancing
• Amdahl’s vs. Gustafson's law
• Superlinear speedup
What you will learn today

• Different levels of parallelism
  • Instruction level parallelism
  • Single Instruction, Multiple Data (SIMD)
  • Multicores

• Cache coherence
Simple CPU

• CPU is a digital circuit – time discretized in cycles.
  • $F = \text{frequency (instructions per second)}$
  • $T_{\text{cycle}} = \frac{1}{F}$ (time of a single cycle)
• Assume: 1 instruction is executed in 1 cycle
• Does it mean $T_e = \frac{\#\text{executed instructions}}{f}$?
• Does the effective Instructions per Cycle (IPC) is always 1?
NO

• Instruction should be interpreted by CPU
  • Takes ~5 cycles

• Memory is slow – results in stalls
  • 100-500 cycles
Pipelining

• Instruction decoding is split into stages
• One instruction is being executed while another is being decoded
• Outcome: \( throughput = \frac{4 \text{ instructions}}{4 \text{ cycles}} \) IPC
  
  • Only if pipeline is full!
Can we improve with more parallelism?

Yes!
Instruction level parallelism (ILP)

• Consider the following code:

1. load  R1, @1000
2. load  R2, @1008
3. add   R1, @1004
4. add   R2, @100C
5. add   R1, R2
6. store R1, @2000

• What does this code do?
• Can it be parallelized?
Independent instructions

• 1 independent of 2
• 3 independent of 4

• We need:
  • Hardware to detect that
  • Hardware to issue multiple instructions
  • Hardware to execute them (maybe out-of-order)

• We get superscalar processor

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>load R1, @1000</td>
</tr>
<tr>
<td>2.</td>
<td>load R2, @1008</td>
</tr>
<tr>
<td>3.</td>
<td>add R1, @1004</td>
</tr>
<tr>
<td>4.</td>
<td>add R2, @100C</td>
</tr>
<tr>
<td>5.</td>
<td>add R1, R2</td>
</tr>
<tr>
<td>6.</td>
<td>store R1, @2000</td>
</tr>
</tbody>
</table>
How does it work?

(Two-way super scalar)

Instruction cycles

0 2 4 6 8

IF: Instruction Fetch
ID: Instruction Decode
OF: Operand Fetch
E: Instruction Execute
WB: Write-back
NA: No Action

load R1, @1000
load R2, @1008
add R1, @1004
add R2, @100C
add R1, R2
store R1, @2000
Bad news: Instruction Level Parallelism (ILP) is limited
Vectorization and VLIW

• Why not let compiler do the job
• Vectorization: processing multiple data with the same instruction (SIMD)

```c
for(int i=0; i < 4; i++){
    c[i]=a[i]+b[i];
}
```

• Very Large Instruction Word (MIMD)
  • Multiple instructions are grouped into one large one
  • Can reduce horizontal waste

```c
C_vec=A_vec+B_vec;
```
How can we reduce vertical waste?

• We will need to understand the memory characteristics.
Memory

• Performance characteristics
• Example: which drop will reach the other side first?
Memory

• Performance characteristics
  • Latency – time between data request and receipt
  • Bandwidth – rate at which data is moved
Why latency is a problem?

- 1GHz processor
  - Memory latency = 100ns.
  - Memory bandwidth = Infinity
  - Assume 1 floating point ADD = 1 cycle (1ns)
- RAW ALU performance: 1 GFLOPs/s

- Observed performance for adding two vectors?
  - We need 3 memory accesses = 300ns per 1 addition
    - => 300 times slower
  - If we parallelize the two reads
    - Then “only” 200 times slower
  - For a vector add, we only pay the latency penalty twice
Why **bandwidth** is a problem?

- 1GHz processor
  - Memory latency = 0 ns
  - Memory bandwidth = 1GB/s
  - Assume 1 floating point ADD = 1 cycle (1ns)
- RAW ALU performance: 1 GFLOPs/s

- Observed performance for adding two vectors?
  - We need 12 bytes per one cycle: 12 GB/s.
  - But we can do only 1GB/s
  - => 12 times slower
Coping with memory limitations

Cache

• Observation 1: small is fast
  • Small cache has very low latency
  • Cache can be placed ON chip
  • Therefore, it has high bandwidth

• Observation 2: most applications reuse data
  • Spatial locality: most accesses are close in space
  • Temporal locality: most accesses to the same data are close in time
Cache performance

• If data in cache (cache hit):
  • Latency = 1 cycle
  • Bandwidth = 100 GB/s

• Otherwise: miss penalty
  • Access next cache level
  • Cache hierarchy follows same principles
  • The last level cache access the memory

• Hit rate = \frac{\#hits}{\#total \ accesses}
  • On what does the hit rate depends?

• Miss rate?
Cache performance example

• Hit rate = 90%
• 20% instructions access memory
• Miss penalty: 100 cycles
• Performance?

• What if hit rate=99%?
Question:
Does cache help for adding two vectors?
NO!

• There is no data reuse.
• Cache is populated upon first access
  • Compulsory miss (slow)
• If the data is not reused – no benefit from cache
• Almost true:
  • Programmed prefetching helps
  • Hardware prefetching also helps
  • => Memory bandwidth better utilized
Question:
Does cache help for matrix product?
Yes!

- $O(n^3)$ computations
- $O(n^2)$ memory
- We read each datum $O(n)$ times

- The trivial implementation is not cache efficient
  - Read more Chapter 2, Grama textbook
Checkpoint

• Pipelines improve CPU performance by hiding instruction handling delay
• Super-scalar out-of-order processors can “automatically” improve the performance by exploiting Instruction Level Parallelism
• Caches reduce latency by assuming spatial and temporal locality
  • Caches are VERY important – lots of research done on cache-efficient algorithms
• What else can we do?
Adding more cores (CMP)

- Chip Multi Processor (CMP)
  - Same chip, multiple units
Adding more processors

• Shared Memory Processors (SMPs)
  • Single physical memory space
  • Each processor/core(s) is on a different chip
Cache coherence problem

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A=0;</td>
</tr>
<tr>
<td></td>
<td>A=1;</td>
</tr>
<tr>
<td></td>
<td>print(A);</td>
</tr>
</tbody>
</table>

Which value will be printed?
Cache coherence protocol

• Must ensure serializibility
  • There must be an equivalent global order of reads and writes that respects (is consistent with) the local order of each of the threads
Cache coherence

- Snoopy cache coherence
  - All caches listen for other caches' reads and writes
  - Write: invalidates values in others
  - Reads: get the latest version from other cache
Cache coherence-2

Multiple copies
All valid

Invalid
Current copy invalid

Shared

Dirty
Current copy valid
Others are invalid

C_read
C_write
flush
read/write
read
write
False sharing

- Cache coherence is managed at cache line granularity (e.g. 64 bytes – i.e. 8 pointers)
- Variables on the same cache line will be invalidated
- May lead to unexpected coherence traffic
Summary

• Snoopy cache coherence:
  • Requires broadcasts
  • There are other approaches

• False sharing: can generate unexpected load
  • Delays memory access
  • Quickly grows with #processors – not scalable