Concurrent and Distributed Programming

Lecture 2
Parallel architectures
Performance of parallel programs

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

- Different levels of parallelism
  - Instruction level parallelism
  - SIMD
  - Multicores
- Cache coherence
CPU is a digital circuit – time discretized in cycles. 1 cycle = $1/F_{\text{frequency}}$.

Assume: 1 instruction is executed in 1 cycle

Does it mean $T_e = \#\text{executed instructions} / F$?
NO

- Instruction should be understood 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

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>OF</th>
<th>E</th>
<th>WB</th>
</tr>
</thead>
</table>

Fetch Inst
Decode Inst
Operand Inst
Execute
Write back
**Outcome:** throughput = 4 inst/4cycles

  *(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

- Can it be parallelized?
Independent instructions

- 1 independent of 2
- 3 independent of 4
- Processor can detect that!
- 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. load R1, @1000</td>
<td>2. load R2, @1008</td>
</tr>
<tr>
<td>3. add R1, @1004</td>
<td>4. add R2, @100C</td>
</tr>
<tr>
<td>5. add R1, R2</td>
<td>6. store R1, @2000</td>
</tr>
</tbody>
</table>
How does it work

*Food for thought – how to cope with loops?*

*Hint – guess..*
Bad news: ILP is limited
Vectorization and VLIW

- Why not let compiler do the job
- Vectorization: processing multiple data with the same instruction (SIMD)
  
  ```
  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
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: 1GFLOPs/s
- Observed performance for adding two vectors?
Why latency is a problem

- 1GHz processor
  - Memory latency = 100ns.
  - Memory bandwidth = Infinity
  - Assume 1 floating point ADD = 1 cycle (1ns)
- RAW ALU performance: 1GFLOPs/s
- Observed performance for adding two vectors?
  - We need 3 memory accesses = 300ns per 1 addition => 300 times slower
Why bandwidth is a problem - 2

- 1GHz processor
  - Memory latency = 0 ns
  - Memory bandwidth = 1GB/s
  - Assume 1 floating point ADD = 1 cycle (1ns)
- RAW ALU performance: 1GFLOPs/s
- Observed performance for adding two vectors?
Why bandwidth is a problem - 2

- 1GHz processor
  - Memory latency = 0 ns
  - Memory bandwidth = 1GB/s
  - Assume 1 floating point ADD = 1 cycle (1ns)
- RAW ALU performance: 1GFLOPs/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: most applications reuse data
  - **Spatial locality**: most accesses are close in space
  - **Temporal locality**: most accesses to the same data are close in time

- Observation 2: small is fast
  - Cache can be placed ON chip
  - Cache has low latency
  - Cache has high bandwidth
Cache performance

- If data in cache (cache hit): latency = 1 cycle, bandwidth = 100s GB/s
- Otherwise – miss penalty
- Hit rate = \(\frac{\text{hits}}{\text{total accesses}}\)
- Miss rate?
- Example: hit rate = 90%, 20% instructions access memory, miss penalty 100 cycles.
- Performance? What if hit rate = 99%?
- Cache hierarchy follows same principles
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
    - 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
- 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
Adding more processors

- Shared Memory Processors (SMPs)
  - Single physical memory space
SMPs vs. Multicores

- SMP – different chips
- Multicores (CMP) – same chip, multiple units
Cache coherence problem

Thread 1
A=1;

Thread 2
A=0;
print(A);

CPU

Memory

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

```
T1: R a,#2; W a,#3
T2: W a,#2; R a,#3
```

```
T1: R a,#2; W a,#3
T2: W a,#2; R a,#3
```
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

- Shared
  - Multiple copies
    - All valid
  - C_read
  - C_write
- Invalid
  - read
  - write
  - C_write
  - flush
  - read/write
- Dirty
  - write
  - C_write
  - read/write

- Current copy invalid
  - Others are invalid
## Cache Coherence -3

<table>
<thead>
<tr>
<th>time ↓</th>
<th>Instruction at proc 0</th>
<th>Instruction at proc 1</th>
<th>Variables’ states at proc 0</th>
<th>Variables’ states at proc 1</th>
<th>States at global mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x=5, S</td>
<td>y=12, S</td>
<td>x=6, D</td>
<td>y=13, D</td>
<td>x=5, I</td>
</tr>
<tr>
<td>x=x+1</td>
<td>x=6, D</td>
<td>y=13, D</td>
<td>x=6, S</td>
<td>y=13, S</td>
<td>x=12, I</td>
</tr>
<tr>
<td>y=y+1</td>
<td>x=6, S</td>
<td>y=13, D</td>
<td>x=6, S</td>
<td>y=13, S</td>
<td>x=6, I</td>
</tr>
<tr>
<td>read y</td>
<td>y=13, S</td>
<td>x=6, S</td>
<td>x=6, S</td>
<td>y=32, D</td>
<td>x=19, S</td>
</tr>
<tr>
<td>read x</td>
<td>x=6, S</td>
<td>y=13, D</td>
<td>x=6, I</td>
<td>y=13, I</td>
<td>x=19, I</td>
</tr>
<tr>
<td>x=x+y</td>
<td>x=19, D</td>
<td>x=6, I</td>
<td>x=6, I</td>
<td>y=13, I</td>
<td>x=19, I</td>
</tr>
<tr>
<td>y=x+y</td>
<td>x=19, S y=13, I</td>
<td>x=19, S y=32, D</td>
<td>x=19, S y=13, I</td>
<td>x=19, I</td>
<td>x=19, I</td>
</tr>
<tr>
<td>x=x+1</td>
<td>x=20, D</td>
<td>y=33, D</td>
<td>x=13, I</td>
<td>y=13, I</td>
<td>x=19, I</td>
</tr>
<tr>
<td>y=y+1</td>
<td>x=20, D</td>
<td>y=33, D</td>
<td>x=13, I</td>
<td>y=13, I</td>
<td>x=19, I</td>
</tr>
</tbody>
</table>
False sharing

- Cache coherence is managed at cache line granularity (e.g. 4 words)
- 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
Performance of parallel programs

More details in Chapter 5 in Graha book

- A parallel system is a combination of a parallel algorithm and an underlying platform

- Intuitive measures
  - Wall clock time
  - Speedup = (Serial time)/(Parallel time)
  - GFLOPs/s = how well the hardware is exploited

- Need more:
  - Scalability: Speedup as a function of #CPUs
  - Efficiency: Speedup/#CPUs
Can we estimate an upper bound on speedup? Amdahl's law

- Sequential component limits the speedup

- Split program serial time $T_{\text{serial}}=1$ into
  - Ideally parallelizable: $A$ (fraction parallel): ideal load balancing, identical speed, no overheads
  - Cannot be parallelized: $1-A$
  - Parallel time $T_{\text{parallel}}=A/#\text{CPUs}+(1-A)$

- Speedup(#CPUs) = $T_{\text{serial}}/T_{\text{parallel}} = \frac{1}{\left(\frac{A}{#\text{CPUs}}+(1-A)\right)}$
Bad news

Source: wikipedia

So - why do we need machines with 1000x CPUs?
Living with Amdahl's law: Gustafson's law

\[ T_{\text{parallel}} = T_{\text{parallel}} \times (A + (1-A)) \]

\[ T_{\text{best serial}} \leq \#\text{CPUs} \times T_{\text{parallel}} \times A + T_{\text{parallel}} \times (1-A) \]

[by simulation, a bound on the best serial program]

\[ \text{Speedup} = \frac{T_{\text{best serial}}}{T_{\text{parallel}}} \leq \#\text{CPUs} \times A + (1-A) = \#\text{CPUs} - (1-A) \times (\#\text{CPUs} - 1) \]

It all depends how good the simulation is

Simulation usually improves with problem size
Amdahl vs. Gustafson – both right

It is all about granularity

- When problem size is fixed granularity diminishes with #CPU
- When granularity diminishes, simulation departs from "bestserial", and Gustafson's upper-bound departs from the actual speedup

Amdahl's law: strong scaling

- Solve same problems faster for fixed problem size and #CPU grows
- If A is fixed, granularity diminishes

Gustafson's law: weak scaling

- Problem size grows: solve larger problems
- Keep the amount of work per CPU when adding more CPUs to keep the granularity fixed
Question

- Can we achieve speedups higher than $\#$CPUs?
- “Superlinear”
Fake superlinear speedup: Serial algorithm does more work

- Parallel algorithm is BFS
- DFS is inefficient for this input
- BFS can be simulated with a serial algorithm
Always use best serial algorithm as a baseline

Example: sorting an array

- Efficient parallel bubble sort takes 40s, serial 150s. Speedup = 150/40?

- **NO.** Serial quicksort runs in 30s. Speedup 0.75!
True superlinear speedup

*Example: cache*

Parallelization results in smaller problem size/CPU => if fits the cache

=> non-linear performance boost!

- Cannot be *efficiently* simulated on a serial machine
- See more examples in the Graha book
Summary

- Parallel performance is a subtle matter
- Need to use a good serial program
- Need to understand the hardware

Amdahl's law: understand the assumptions!!!

Too pessimistic
- Fraction parallel independent of #CPUs
- Assumes fixed problem size

Too optimistic
- Assumes perfect load balancing, no idling, equal CPUs speed

See more material: refs/ on website and book