Conclusions

- The multicore era is already here
- Cost of parallelism is dropping dramatically
  - Cache/Thread is dropping
  - Memory bandwidth/thread is dropping
- The introduction of multicore processors will have profound impact on computational software

“For high-performance computing (HPC) applications, multicore processors introduces an additional layer of complexity, which will force users to go through a phase change in how they address parallelism or risk being left behind.”

[IDC presentation on HPC at International Supercomputing Conference, ICS07, Dresden, June 26-29 2007]
Why multiple cores/threads? (even in your laptops)

- Instruction level parallelism is running out
- Wire delay is hurting
- Power dissipation is hurting
- The memory latency/bandwidth bottleneck is becoming even worse
Focus: Performance

Create and explore:

1. Parallelism
   - Instruction level parallelism (ILP)
   - In some cases: Parallelism at the program/algorithm level ("Parallel computing")

2. Locality of data
   - Caches
   - Temporal locality
   - Cache blocks/lines: Spatial locality
Old Trend 1: Deeper pipelines (exploring ILP)
Old Trend 2: Wider pipelines (exploring more ILP)
Old Trend 3: Deeper memory hierarchy
(exploring locality of data)
Are we hitting the wall now?

"Moore's law"

Possible path, but requires a paradigm shift

Business as usual...

Humm, transistors can be made smaller and faster, but there are other problems
Classical microprocessors: Whatever it takes to run **one** program fast

- Faster clocks → Deep pipelines
- Superscalar Pipelines
- Branch Prediction
- Out-of-Order Execution
- Trace Cache
- Speculation
- Predicate Execution
- Advanced Load Address Table
- Return Address Stack
- ...

Bad News #1: Already explored most ILP
#2: Future technology

Quantitative data and trends according to V. Agarwal et al., ISCA 2000
Based on SIA (Semiconductor Industry Association) prediction, 1999
#2: How much of the chip area can you reach in one cycle?

Bad News #2: Looong wire delay → slow CPUs
Bad News #3: Mem latency/bandwidth is the bottleneck

\[
A = B + C: \quad \text{Latency} \quad \begin{align*}
\text{Read } B & : & 0.3 - 100 \text{ ns} \\
\text{Read } C & : & 0.3 - 100 \text{ ns} \\
\text{Add } B \& C & : & 0.3 \text{ ns}
\end{align*}
\]
Bad News #4: Power is the limit

- Power consumption is the bottleneck
  - Cooling servers is hard
  - Battery lifetime for mobile computers
  - Energy is money

- Dissipated effect is proportional to
  - $\sim$ Frequency
  - $\sim$ Voltage$^2$
1. **Why multiple threads/cores?**
   - Old Trends
   - Bad news
   - Solutions?

2. **Introducing concurrency**
   - Scenario
   - Definitions
   - Amdhal’s law

3. **Can we trust the hardware?**
Bad News #1: Not enough ILP
→ feed one CPU with instr. from many threads
Simultaneous multithreading
Bad News #2: Wire delay
→ Multiple small CPUs with private L1$
Bad News #2: Wire delay

→ Multiple small CPUs with private L1$
Bad News #3: Memory latency/bandwidth

→ memory accesses from many threads
Bad News #4: Power consumption
→ Lower the frequency → lower voltage

\[ P_{\text{dyn}} = C \times f \times V^2 \approx \text{area} \times \text{freq} \times \text{voltage}^2 \]

<table>
<thead>
<tr>
<th>CPU freq=f</th>
<th>VS.</th>
<th>CPU freq=f/2</th>
<th>CPU freq=f/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{\text{dyn}}(C, f, V) &lt; CfV^2 )</td>
<td>( P_{\text{dyn}}(2C, f/2, &lt; V) &lt; CfV^2 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU freq=f</th>
<th>VS.</th>
<th>freq=f/2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_{\text{dyn}}(C, f, V) &lt; CfV^2 )</td>
<td>( P_{\text{dyn}}(C, f/2, &lt; V) &lt; \frac{1}{2} CfV^2 )</td>
<td></td>
</tr>
</tbody>
</table>
Solving all the problems (?): Exploring thread parallelism

#1 Running out of ILP
→ feed one CPU with instr. from many threads

#2 Wire delay is starting to hurt
→ Multiple small CPUs with private caches

#3 Memory is the bottleneck
→ memory accesses from many threads

#4 Power is the limit
→ Lower the frequency → lower voltage
In all computers very soon...

Multicore processors, probably also with simultaneous multithreading
Scenario

Several cars want to drive from point A to point B.

They can compete for space on the same road and end up either:
- following each other
- or competing for positions (and having accidents!).

Or they could drive in parallel lanes, thus arriving at about the same time without getting in each other’s way.

Or they could travel different routes, using separate roads.
Scenario

Several cars want to drive from point A to point B.

Sequential Programming

They can compete for space on the same road and end up either:
- following each other
- or competing for positions (and having accidents!).

Parallel Programming

Or they could drive in parallel lanes, thus arriving at about the same time without getting in each other's way.

Distributed Programming

Or they could travel different routes, using separate roads.
Concurrent Program

- \(2^+\) processes working together to perform a task.
- Each process is a sequential program
  (= sequence of statements executed one after another)
- *Single thread of control* vs *multiple thread of control*

Communication
- Shared Variables
- Message Passing

Synchronization
- Mutual Exclusion
- Condition Synchronization
Wanna write a concurrent program?

- What kinds of processes?
- How many processes?
- How should they interact?

Correctness

Ensure that processes interaction is properly synchronized

- **Mutual Exclusion**
  Ensuring the critical sections of statements do not execute at the same time

- **Condition Synchronization**
  Delaying a process until a given condition is true

Our focus: imperative programs and asynchronous execution
Amdhal’s law

- \( P \) is the fraction of a calculation that can be parallelized
- \((1 - P)\) is the fraction that is sequential (i.e. cannot benefit from parallelization)
- \( N \) processors

\[
\text{speedup} = \frac{(1-P)+P}{(1-P)+P/N}
\]
\[
\text{maximum speedup} = \frac{1}{1-P}
\]

Example

If \( P = 90\% \) \( \Rightarrow \) max speedup of 10

no matter how large the value of \( N \) used (ie \( N \to \infty \))
## Single-Processor machine

<table>
<thead>
<tr>
<th></th>
<th>CPU</th>
<th>Cache</th>
<th>Mem</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Level 1</td>
<td>Level 2</td>
<td></td>
</tr>
<tr>
<td>2000:</td>
<td>1ns</td>
<td>3ns</td>
<td>10ns</td>
<td>150ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5 000 000ns</td>
</tr>
<tr>
<td>1982:</td>
<td>200ns</td>
<td>200ns</td>
<td>200ns</td>
<td>10 000 000ns</td>
</tr>
</tbody>
</table>
Memory Hierarchy

- Main Memory
  - Level 2 cache
  - Level 1 cache
  - CPU
Why do we miss in the cache?

- **Compulsory miss**
  - Touching the data for the first time

- **Capacity miss**
  - The cache is too small

- **Conflict miss**
  - Non-ideal cache implementation (data hash to the same cache line)
Locality

- Temporal locality
- Spatial locality

Inner loop stepping through an array

A, B, C, A+1, B, C, A+2, B, C,

spatial  temporal
MultiProcessor world - Taxonomy

- SIMD
- MIMD
- Message Passing
- Shared Memory
- Fine-grained
- Coarse-grained
- UMA
- NUMA
- COMA
Shared-Memory Multiprocessors
Programming Model

Shared Memory

$ Thread

Thread

Thread

Thread

Thread

Thread

Thread

Thread
Cache coherency

A: Thread
  Read A
  Read A
  ... 
  Read A
  Write A

B: Thread
  Read A
  Read A
  ... 
  Read A
  Read B

Shared Memory
Summing up Coherence

There can be many copies of a datum, but only one value.

There is a single global order of value changes to each datum.
Memory Ordering

- The **coherence** defines a per-datum order of value changes.
- The **memory model** defines the order of value changes for all the data.

What ordering does the memory system guarantees? “Contract” between the HW and the SW developers

*Without it, we can’t say much about the result of a parallel execution*
What order for these threads?

A’ denotes a modified value to the datum at address A

Thread 1

LD A
ST B’
LD C
ST D’
LD E
...
...

Thread 2

ST A’
LD B’
ST C’
LD D
ST E’
...
...
Other possible orders?

Thread 1
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...

Thread 2
- ST A'
- LD B'
- ST C'
- LD D
- ST E'
- ...
- ...

Thread 1
- LD A
- ST B'
- LD C
- ST D'
- LD E
- ...
- ...

Thread 2
- ST A'
- LD B'
- ST C'
- LD D
- ST E'
- ...
- ...
Memory model flavors

- **Sequentially Consistent**: Programmer’s intuition
- **Total Store Order**: Almost Programmer’s intuition
- **Weak/Release Consistency**: No guaranty
Dekker’s algorithm

Initially $A = 0, B = 0$

“fork”

Does the write become globally visible before the read is performed?

$A := 1$
if($B == 0$) print(“$A$ wins”);

$B := 1$
if($A == 0$) print(“$B$ wins”);

Can both $A$ and $B$ win?

Left: The read (ie, test if $B == 0$) can bypass the store ($A := 1$)
Right: The read (ie, test if $A == 0$) can bypass the store ($B := 1$)
⇒ Both loads can be performed before any of the stores
⇒ Yes, it is possible that both win!
Dekker’s algorithm for Total Store Order

Initially $A=0, B=0$

“fork”

Does the write become globally visible before the read is performed?

$A := 1$
Membar #StoreLoad;
if($B==0$)print(“$A$ wins”);

$B := 1$
Membar #StoreLoad;
if($A==0$)print(“$B$ wins”);

Can both $A$ and $B$ win?

Membar: the read is started after all previous stores have been “globally ordered”
⇒ Behaves like a sequentially consistent machine
⇒ No, they won’t both win. Good job Mister Programmer!
Dekker’s algorithm, in general

Initially $A = 0, B = 0$

“fork”

\[
\begin{align*}
A &:= 1 \\
\text{if} (B == 0) \text{print}("A wins");
\end{align*}
\]

\[
\begin{align*}
B &:= 1 \\
\text{if} (A == 0) \text{print}("B wins");
\end{align*}
\]

Can both $A$ and $B$ win?

The answer depends on the memory model

Remember? ...

**Contract** between the HW and SW developers.
Memory Model is a tricky issue
New issues

- Compulsory miss
- Capacity miss
- Conflict miss

- Communication miss
  - Cache-to-cache transfer

- False-sharing
  - Side-effect from large cache lines

- What about the compiler?
  - Code reordering? `volatile` keyword in C...
Performance $\Rightarrow$ Use of Cache
Memory hierarchy $\Rightarrow$ Consistency problems

To get maximal performance on a given machine,
the programmer has to know about the characteristics of the memory system and has to write programs to account them
Distributed Memory Architecture

- Communication through Message Passing
- Own cache, but memory not shared
  ⇒ No coherency problems
Isn’t a CMP just a SMP on a chip?

Symmetric MultiProcessor, SMP

<table>
<thead>
<tr>
<th>L2</th>
<th>L2</th>
<th>... 32 ...</th>
<th>L2</th>
<th>L1</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L1</td>
<td></td>
<td>L1</td>
<td></td>
<td>CPU</td>
</tr>
</tbody>
</table>

CMP

<table>
<thead>
<tr>
<th>L2</th>
<th>L1</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td>8...</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>
Cost of communication?
Impact on Algorithms

For performance, we need to understand the interaction between algorithms and architecture.

The rules have changed

We need to question old algorithms and results!
Criteria for algorithm design

- **Pre-CMP:**
  - Communication is expensive: Minimize communication
  - Data locality is important
  - Maximize scalability for large-scale applications

- **Within a CMP chip today:**
  - (On-chip) communication is almost to free
  - Data locality is even more important
  - (SMT may help by hiding some poor locality)
  - Scalability to 2-32 threads

- **In a multi-CMP system tomorrow:**
  - Communication is sometimes almost free (on-chip), sometimes (very) expensive (between chips)
  - Data locality (minimizing of-chip references) is a key to efficiency
  - “Hierarchical scalability”