Multi-Core & Parallel Processing

I’ve gotta spend at least 10 hours studying for the Comp 411 final!

I’m going to study with 9 friends... we’ll be done in an hour.

Chapter 7
TIPs Anyone?

MIPS = \frac{\text{Clock Frequency (in MHz)}}{\text{Clocks per Instruction}}

Mega – 10^6  Giga – 10^9  Tera – 10^{12}  Peta – 10^{15}

Light travels about 1 ft / 10^{-9} secs in free space.
A Tera-Hertz uniprocessor could have no clock-to-clock path longer than 300 microns (thickness of a hair)…

We already know of problems that require greater than a TIP
(Simulations of weather, weapons, brains)

I guess that means that there are 10^{12} microphones in a Megaphone?
Driving Down the Denominator

Techniques for increasing parallelism:

**Pipelining** – reasonable for a small number of stages (5-10), after that bypassing and stalls become unmanageable.

**Superscalar** – replicate data paths and design control logic to discover parallelism in traditional programs.

**Explicit parallelism** – must learn how to write programs that run on multiple CPUs.
Superscalar Parallelism

- Multiple Functional Units (ALUs, Addr units, etc)
- Multiple instruction dispatch
- Dynamic Pipeline Scheduling
- Speculative execution

Popular 5-10 years ago – but the end is near!
Explicit Parallelism

Three key aspects of Parallel computing:
Control, Communications, and types of processing elements

<table>
<thead>
<tr>
<th>Control</th>
<th>Communication</th>
<th>Processing Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unified</td>
<td>Shared Memory</td>
<td>Homogeneous</td>
</tr>
<tr>
<td>Distributed</td>
<td>Message Passing</td>
<td>Heterogeneous</td>
</tr>
</tbody>
</table>

Decoding the Parallel Processor Alphabet Soup:

**SIMD** - Single-Instruction-Multiple-Data
Unified control, Homogeneous processing elements

**VLIW** - Very-Long-Instruction-Word
Unified control, Heterogeneous processing elements

**MIMD** - Multiple-Instruction-Multiple-Data
Distributed control, Message Passing

**SMP** – Symmetric Multi-Processor
Distributed control, Shared memory, Homogenous PEs
SIMD Processing

Each datapath has its own local data (Register File)
All data paths execute the same instruction

Conditional branching is difficult…
(What if only one CPU has R1 == $0?)

Conditional operations are more common in SIMD machines
if (flag1) Rc = Ra <op> Rb

Global ANDing or ORing of flag registers are used for high-level control
SIMD Coprocessing Units

SIMD data path added to a traditional CPU core
Register-only operands
Core CPU handles memory traffic
Partitionable Datapaths for variable-sized “PACKED OPERANDS”
SIMD Coprocessing Units

Intel “MMX”
Multimedia Extensions

SIMD data path added to a traditional CPU core
Register-only operands
Core CPU handles memory traffic
Partitionable Datapaths for variable-sized “PACKED OPERANDS”
SIMD Coprocessing Units

SIMD data path added to a traditional CPU core
Register-only operands
Core CPU manages memory traffic
Partitionable Datapaths for variable-sized
“PACKED OPERANDS”

Intel “MMX” Multimedia Extensions

Nice data size for:
Graphics,
Signal Processing,
Multimedia Apps,
etc.
SIMD Coprocessing Units

Intel “MMX” Multimedia Extensions

SIMD data path added to a traditional CPU core
Register-only operands
Core CPU manages memory traffic
Partitionable Datapaths for variable-sized “PACKED OPERANDS”

MMX instructions:
- PADDB - add bytes
- PADDW - add 16-bit words
- PADDD - add 32-bit words (unsigned & w/saturation)
- PSUB{B,W,D} – subtract
- PMULTLW – multiply low
- PMULTHW – multiply high
- PMADDW – multiply & add
- PACK –
- UNPACK –
- PAND –
- POR -
VLIW Variant of SIMD Parallelism

A single-WIDE instruction controls multiple heterogeneous datapaths.

Exposes parallelism to compiler (S/W vs. H/W)
MIMD: Multi-CPU Architecture

- **Reaction to Superscalar Approach**
  - Diminishing returns for H/W to find instruction-level parallelism
  - Improving Superscalar H/W becomes more and more complex
  - Give up, and let the S/W folks figure it out

- **Multiple CPUs (each with its own a PC/program)**

- **H/W focuses on communication**
  - Crossbars (switches to share multiple buses)
  - Meshes (point-to-point store-and-forward communication)
  - Shared Caches and memory interfaces (further taxing a known bottleneck)

- **S/W focuses on partitioning of data & algorithms**
MIMD Example - Shared memory

SMP – Symmetric Multiprocessors
All processors are identical and share a common main memory
Leverages existing CPU architectures / designs
Easy to migrate “Processes” to “Processors”
Share data and program
Communicate through shared memory
Easy upgrades (more CPUs)

Problems:
Scalability
Synchronization

Main Memory

Mini MIPS

Mini MIPS

Mini MIPS

Mini MIPS

Mini MIPS

Mini MIPS

Main Memory
Symmetric Multiprocessor Fantasies

If one processor is good, N processors are GREAT:

\[ P_1 \quad P_2 \quad \ldots \quad P_N \]

Shared Main Memory

IDEA:

- Run N processes, each on its OWN processor!
- Processors compete for bus mastership, memory access
- Bus SERIALIZES memory operations (via arbitration for mastership)

PROBLEM:

The Bus quickly becomes the BOTTLENECK
Multiprocessor with Caches

But, we’ve seen this problem before. The solution, add CACHES.

Consider the following trivial processes running on $P_1$ and $P_2$:

**Program A**

```
x = 3;
print(y);
```

**Program B**

```
y = 4;
print(x);
```
What are the Possible Outcomes?

Simulation of two processors, each with a write-back cache

**Processor A**

\[
x = 3; \\
\text{print}(y);
\]

\[
_A: x = 1 \\
y = 2
\]

**Processor B**

\[
y = 4; \\
\text{print}(x);
\]

\[
_B: x = 1 \\
y = 2
\]

All plausible interleaved execution sequences:

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>A prints</th>
<th>B prints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=3; print(y); y=4; print(x);</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x=3; y=4; print(y); print(x);</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>x=3; y=4; print(x); print(y);</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>y=4; x=3; print(x); print(y);</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>y=4; x=3; print(y); print(x);</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>y=4; print(x); x=3; print(y);</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Modifications made by one CPU aren’t seen by the others until the corresponding cache line is replaced.
Compare to Uniprocessor Outcome

But, what are the possible outcomes if we ran Process A and Process B on our single “timed-shared” processor from last lecture?

**“Process” A**

\[
x = 3;
\]

print(y);

**“Process” B**

\[
y = 4;
\]

print(x);

Plausible Uniprocessor execution sequences:

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>A prints</th>
<th>B prints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=3; print(y); y=4; print(x);</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>x=3; y=4; print(y); print(x);</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>x=3; y=4; print(x); print(y);</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>y=4; x=3; print(x); print(y);</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>y=4; x=3; print(y); print(x);</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>y=4; print(x); x=3; print(y);</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice that the 2 processor outcome “2, 1” does not even appear in our list!
Parallel Sequential Consistency

Semantic constraint:

Result of executing N parallel programs should correspond to some interleaved execution on a single processor.

Process A

\[ \begin{align*}
  x &= 3; \\
  \text{print}(y); \\
\end{align*} \]

Process B

\[ \begin{align*}
  y &= 4; \\
  \text{print}(x); \\
\end{align*} \]

Possible printed values: 2, 3; 4, 3; 4, 1.
(each corresponds to at least one interleaved execution)

IMPOSSIBLE printed values: 2, 1
(corresponds to NO valid interleaved execution).

Weren’t caches supposed to be invisible to programs?
Cache Incoherence

PROBLEM: “stale” values in cache ...

$P_1$: $x=3$
  
  $y=2$

$P_2$: $x=1$
  
  $y=4$

The problem is not that memory has stale values, but that other caches may!

$P_2$:

\[ x = 3, y = 4 \]

Process A

\[
\begin{align*}
  x &= 3; \\
  \text{print}(y); \\
\end{align*}
\]

Process B

\[
\begin{align*}
  y &= 4; \\
  \text{print}(x); \\
\end{align*}
\]

Q: How does B know that A has changed the value of x?

The problem is not that memory has stale values, but that other caches may!
Cache Coherence Solutions

Problem: A writes data into shared memory; B still sees “stale” cached value.

Solutions:

1. Don’t cache shared Read/Write pages.
   COST: Longer access time to shared memory.

2. Attach cache to shared memory, not to processors...
   ... share the cache as well as the memory!
   
   COSTS:
   1. **Adds Bus Contention**
   2. **Reduces Locality**

3. Make caches talk to each other, maintain a consistent story.
“Snoopy” Caches

IDEA:

- $P_1$ writes 3 into $x$; write-thru cache causes bus transaction.
- $P_2$, snooping, sees transaction on bus and either INVALIDATES or UPDATES its cached copy of $x$.

MUST WE use a write-thru strategy? (slows down everyone on writes)
Coherency w/ Write Back

IDEA:

- Various caches can have
  - Multiple SHARED read-only copies; OR
  - One UNSHARED exclusive-access read-write copy.
- Keep STATE of each cache line in extra “tag-like” bits (i.e. Valid, Dirty)
- Add bus protocols -- “messages” -- to allow caches to maintain consistent state
## Coherent Cache States

Two-bit STATE in cache line encodes one of M, E, S, I states ("MESI" cache):

- **INVALID**: cache line unused.
- **SHARED ACCESS**: read-only, valid, not dirty. Shared with other read-only copies elsewhere. Must invalidate other copies before writing.
- **EXCLUSIVE**: exclusive copy, not dirty. On write becomes modified.
- **MODIFIED**: exclusive access; read-write, valid, dirty. Must be written back to memory eventually; meanwhile, can be written or read by local processor.

### 4-state FSM for each cache line!

<table>
<thead>
<tr>
<th>Current state</th>
<th>Read Hit</th>
<th>Read Miss, Snoop Hit</th>
<th>Read Miss, Snoop Miss</th>
<th>Write Hit</th>
<th>Write Miss</th>
<th>Snoop for Read</th>
<th>Snoop for Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Modified</td>
<td>Invalid (Wr-Back)</td>
<td>Invalid (Wr-Back)</td>
<td>Modified</td>
<td>Invalid (Wr-Back)</td>
<td>Shared (Push)</td>
<td>Invalid (Push)</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Exclusive Invalid</td>
<td>Invalid</td>
<td>Modified (Invalidate)</td>
<td>X</td>
<td>Modified (Fill-Inv)</td>
<td>Shared</td>
<td>Invalid</td>
</tr>
<tr>
<td>Shared</td>
<td>Shared</td>
<td>Invalid</td>
<td>Invalid</td>
<td>Modified</td>
<td>Invalid</td>
<td>Shared</td>
<td>Invalid</td>
</tr>
<tr>
<td>Invalid</td>
<td>X</td>
<td>Shared (Fill)</td>
<td>Exclusive (Fill)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

4-state FSM for each cache line!

(FREE!!: Can redefine VALID and DIRTY bits)
MERSI Examples

Local WRITE request hits cache line in \textit{Shared} state:

- Send INVALIDATE message forcing other caches to I states
- Change to \textit{Modified} state, proceed with write.

External Snoop READ hits cache line in \textit{Modified} state:

- Write back cache line
- Change to \textit{Shared} state

\text{Mem}[0xc411] = 3


Sequential Inconsistency

Plausible sequence of events:
• A writes 3 into x, sends INVALIDATE message.
• B writes 4 into y, sends INVALIDATE message.
• A reads 2 from y, prints it...
• B reads 1 from y, prints it...
• A, B each receive respective INVALIDATE messages.

FIX: Wait for INVALIDATE messages to be acknowledged before proceeding with a subsequent reads.

COST: Loss of performance (writes stall reads)… must provide for fast invalidates

Process A
\[
\begin{align*}
    &x = 3; \\
    &\text{print}(y); \\
\end{align*}
\]

Shared Memory
\[
\begin{align*}
    &\text{int } x=1, y=2; \\
\end{align*}
\]

Process B
\[
\begin{align*}
    &y = 4; \\
    &\text{print}(x); \\
\end{align*}
\]
Who Needs Parallel Sequential Consistency, Anyway?

ALTERNATIVE MEMORY SEMANTICS:

“WEAK” consistency

EASIER GOAL: Memory operations from each processor appear to be performed in order issued by that processor;

Memory operations from different processors may overlap in arbitrary ways (not necessarily consistent with any interleaving).

COMMON APPROACH:

• Weak consistency, by default;

• MEMORY BARRIER instructions: stalls processor until all previous memory operations have completed.
“Dusty Deck” Problem

How do we make our old sequential programs run on parallel machines? After all, what’s easier, designing new H/W or rewriting all our S/W?

Programs have inertia. Familiar languages, S/W engineering practice reinforce “Sequential Programming Semantics”

By treating PROCESSES or THREADS as a programming constructs… and by assigning each process to a separate processor… we can take advantage of some parallelism.
Programming the Beast

Comp 411 (circa 2012):

```c
int factorial(int n) {
    if (n > 0)
        return n*fact(n-1);
    else
        return 1;
}
```

Calls factorial() only n times

Runs in $O(N)$ time

Comp 411 (circa 2022):

```c
int factorial(int n) {
    return facthelp(1, n);
}
```

```c
parallel int facthelp(int from, int to) {
    int mid;
    if (from >= to) return from;
    mid = (from + to)/2;
    return (facthelp(from,mid)*facthelp(mid+1,to));
}
```

```plaintext
{1, 2, 3, 4, 5, 6, 7, 8}
```

Calls facthelp() $2n - 1$ times
(nodes in a binary tree with $n$ leafs).

Runs in $O(\log_2(N))$ time
(on $N$ processors)
Parallel Processing Summary

Prospects for future CPU architectures:
  Pipelining - Well understood, but mined-out
  Superscalar - Nearing its practical limits
  SIMD - Limited use for special applications
  VLIW - Returns controls to S/W. The future?

Prospects for future Computer System architectures:
  SMP - Limited scalability. Harder than it appears.
  MIMD/message-passing - It’s been the future for over 20 years now. How to program?

WHAT ABOUT THE FUTURE?
  New BOUNDARIES, New PROBLEMS