Four CPU cores sharing Last Level Cache and DRAM main memory

When does it make sense to build systems where many CPUs share a memory system?

And how do we build and program them?

What could we do instead?
Distributed computing

Memory system private to one CPU

Private address spaces, operating system images.

Application-level communication over switch fabric (300\(\mu\)s latency, 100 MB/sec BW)
CPUs share lower level of memory system, and I/O.

Common address space, one operating system image.

Communication occurs through the memory system (100ns latency, 20 GB/s bandwidth)
<table>
<thead>
<tr>
<th>Distributed</th>
<th>Shared-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory system private to one CPU</td>
<td>CPUs share lower levels of memory system, and I/O.</td>
</tr>
<tr>
<td>Private address spaces, operating system images.</td>
<td>Common address space, one operating system image.</td>
</tr>
<tr>
<td>Application-level communication over switch fabric (300µs latency, 100 MB/sec BW)</td>
<td>Communication occurs through the memory system (100ns latency, 20 GB/s bandwidth)</td>
</tr>
</tbody>
</table>
Oracle M6-32

- Fully scaled configuration
- 32 sockets
- 384 cores
- 3072 threads
- 32 TB DRAM [1024 DIMMs]
- 150 ns latency
- 1.4 TB/s memory BW
- 3 Million USD
At $3M, its price undercuts the market leader.

However, the Oracle database you would likely buy to run on the M6-32 costs “extra”.
Google: Warehouse Scale Computer

“Local Rack” has the same machine-room footprint as the Oracle M6-32, and a similar number of cores (320). But much less RAM (1 TB vs. 32 TB).

Google's cost is much less.

“Local Rack” @ dell.com

$150,000

Local Rack

2 servers per “rack unit” (1U)

One Server

DRAM: 16 GB, 100 ns, 20 GB/s
Disk: 2 TB, 10 ms, 200 MB/s
Flash: 128 GB, 100 us, 1 GB/s

2 sockets/server,
2 cores/socket.

Local Rack (80 servers)

DRAM: 1 TB, 300 us, 100 MB/s
Disk: 160 TB, 11 ms, 100 MB/s
Flash: 20 TB, 400 us, 100 MB/s

30$ / port GigE network switch
<table>
<thead>
<tr>
<th>Distributed</th>
<th>Shared-memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>+++ Can scale beyond 10,000 cores in a cost-effective way.</td>
<td>--- Big cost premium above 4 sockets. Scaling limit: 64 sockets.</td>
</tr>
<tr>
<td>--- Applications written to be resilient to hardware failures and “long-tail” latencies.</td>
<td>+++ OS hides hardware failures from application. Latency model is simple, predictable.</td>
</tr>
</tbody>
</table>

For problems too big for the largest shared-memory machine, distributed is the only option. And if costs are a factor, economics push you to distributed if you need more than 4 sockets.
Common Google configuration: “Leaf nodes” are 2 socket, 2 cores per socket, shared memory ...

From: Barroso, Clidaras, and Hölzle, “The Datacenter as a Computer”
Why small shared memory machines as leaf nodes?

Case study: Virtual machines as a service.

Compare a 4-core machine with 16 GB RAM with 4 single-core machines with 4 GB.

(1) Shared-memory machine lets us rent a VM with 16 GB of RAM.

(2) Host operating system resource usage amortized over more guest VMs.
Why small shared memory machines as leaf nodes?

Case study: Application coded to use parallelism.

Compare a 4-core machine with 16 GB RAM with 4 single-core machines with 4 GB.

Application may be able to take advantage of 1000X latency and 100X bandwidth for communication between threads on the same node.
This week: Small Shared-Memory Machines

Coherency and Consistency
Programmer’s view of shared memory.

CPU multi-threading
Keeps memory system busy.

Crossbars and Rings
How to do on-chip sharing.

Concurrent requests
Interfaces that don’t stall.

Coherency Protocols
Building coherent caches.
The Memory Model
Coherency: The Programmer’s View

CPU0:
\[
\text{LW R2, 16(R0)}
\]

CPU1:
\[
\text{LW R2, 16(R0)}
\]

CPU1:
\[
\text{SW R0, 16(R0)}
\]

View of memory no longer coherent.

Loads of location 16 from CPU0 and CPU1 see different values!

Write-through caches

Coherent architectures never let this happen.
Memory Ordering
How 2 threads share a queue ...

We begin with an empty queue ...

Thread 1 (T1) adds data to the tail of the queue.
“Producer” thread

Thread 2 (T2) takes data from the head of the queue.
“Consumer” thread
Producer adding $x$ to the queue ...

**Before:**

```
Words in Memory
```

**T1 code (producer)**

```
ORI R1, R0, xval ; Load x value into R1
LW R2, tail(R0) ; Load tail pointer into R2
SW R1, 0(R2) ; Store x into queue
ADDI R2, R2, 4 ; Shift tail by one word
SW R2 0(tail) ; Update tail memory addr
```

**After:**

```
Words in Memory
```

```
Producer adding y to the queue ...

T1 code (producer)

Before:

Tail

Head

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Words in Memory

Higher Address Numbers

Tail

Head

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y</td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

Words in Memory

Higher Address Numbers

ORI R1, R0, yval ; Load y value into R1
LW R2, tail(R0) ; Load tail pointer into R2
SW R1, 0(R2) ; Store y into queue
ADDI R2, R2, 4 ; Shift tail by one word
SW R2 0(tail) ; Update tail memory addr
Consumer reading the queue ...

Before:

Tail  Head

|   | y    | x    |   |

Words in Memory

LW R3, head(R0) ; Load head pointer into R3
spin: LW R4, tail(R0) ; Load tail pointer into R4
BEQ R4, R3, spin ; If queue empty, wait
LW R5, 0(R3) ; Read x from queue into R5
ADDI R3, R3, 4 ; Shift head by one word
SW R3 head(R0) ; Update head pointer

T2 code (consumer)

After:

Tail  Head

|   | y |

Words in Memory

Higher Address Numbers
What can go wrong? (single-threaded LW/SW “contract”)

What if order is 2, 3, 4, 1? Then, x is read before it is written! The CPU running T1 has no way to know its bad to delay 1!
Leslie Lamport: Sequential Consistency

Sequential Consistency: As if each thread takes turns executing, and instructions in each thread execute in program order.

<table>
<thead>
<tr>
<th>T1 code (producer)</th>
<th>T2 code (consumer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORI R1, R0, x</td>
<td>LW R3, head(R0)</td>
</tr>
<tr>
<td>LW R2, tail(R0)</td>
<td>spin: LW R4, tail(R0)</td>
</tr>
<tr>
<td>SW R1, 0(R2) 1</td>
<td>BEQ R4, R3, spin</td>
</tr>
<tr>
<td>ADDI R2, R2, 4</td>
<td>LW R5, 0(R3) 4</td>
</tr>
<tr>
<td>SW R2 0(tail) 2</td>
<td>ADDI R3, R3, 4</td>
</tr>
<tr>
<td></td>
<td>SW R3 head(R0)</td>
</tr>
</tbody>
</table>

Load x value into R1
Load queue tail into R2
Store x into queue
Shift tail by one word
Update tail memory addr
Load queue head into R3
Load queue tail into R4
If queue empty, wait
Read x from queue into R5
Shift head by one word
Update head memory addr

Sequentially Consistent: 1, 2, 3, 4 or 1, 3, 2, 4 ... but not 2, 3, 1, 4 or 2, 3, 4, 1!
Sequential Consistent architectures get the right answer, but give up many optimizations.
Efficient alternative: Memory barriers

In the general case, machine is **not** sequentially consistent.

When needed, a **memory barrier** may be added to the program (a **fence**).

All memory operations **before** fence complete, then memory operations **after** the fence begin.

```plaintext
ORI R1, R0, x    ;
LW R2, tail(R0)  ;
SW R1, 0(R2)     ; 1
MEMBAR
ADDI R2, R2, 4   ;
SW R2 0(tail)    ; 2
```

Ensures 1 completes before 2 takes effect.

**MEMBAR** is expensive, but you only pay for it when you use it.

Many **MEMBAR** variations for efficiency (versions that only effect loads or stores, certain memory regions, etc).
Producer/consumer memory fences

**T1 code (producer)**

- ORI R1, R0, x ; Load x value into R1
- LW R2, tail(R0) ; Load queue tail into R2
- SW R1, 0(R2) 1 ; Store x into queue
- MEMBAR
- ADDI R2, R2, 4 ; Shift tail by one word
- SW R2 0(tail) 2 ; Update tail memory addr

**T2 code (consumer)**

- LW R3, head(R0) ; Load queue head into R3
- spin: LW R4, tail(R0) 3 ; Load queue tail into R4
- BEQ R4, R3, spin ; If queue empty, wait
- MEMBAR
- LW R5, 0(R3) 4 ; Read x from queue into R5
- ADDI R3, R3, 4 ; Shift head by one word
- SW R3 head(R0) ; Update head memory addr

Ensures $1$ happens before $2$, and $3$ happens before $4$. 

Higher Addresses

Produce: | Head |
---|---
Tail | Produce:

Consume: | Head |
---|---
Tail | Consume:
Break
Sharing Write Access
One producer, two consumers ...

**T1 code (producer)**

- ORI R1, R0, x ; Load x value into R1
- LW R2, tail(R0) ; Load queue tail into R2
- SW R1, 0(R2) ; Store x into queue
- ADDI R2, R2, 4 ; Shift tail by one word
- SW R2 0(tail) ; Update tail memory addr

**T2 & T3 (2 copies of consumer thread)**

- LW R3, head(R0) ; Load queue head into R3
- spin: LW R4, tail(R0) ; Load queue tail into R4
- BEQ R4, R3, spin ; If queue empty, wait
- LW R5, 0(R3) ; Read x from queue into R5
- ADDI R3, R3, 4 ; Shift head by one word
- SW R3 head(R0) ; Update head memory addr

Critical section: T2 and T3 must take turns running red code.
Abstraction: Semaphores (Dijkstra, 1965)

Semaphore: unsigned int s

s is initialized to the number of threads permitted in the critical section at once (in our example, 1).

P(s): If s > 0, s-- and return. Otherwise, sleep. When woken do s-- and return.

V(s): Do s++, awaken one sleeping process, return.

Appears in all threads (initial s = 1):

P(s);
  critical section (s=0)
V(s);

When awake, V(s) and P(s) are atomic: no interruptions, with exclusive access to s.
Spin-Lock Semaphores: Test and Set

An example atomic read-modify-write ISA instruction:

\[
\text{Test\&Set}(m, R) \\
R = M[m]; \\
\text{if } (R == 0) \text{ then } M[m]=1;
\]

Note: With Test\&Set(), the \( M[m]=1 \) state corresponds to last slide's \( s=0 \) state!

Critical section (consumer)

P: \hspace{1cm} \text{Test\&Set R6, mutex(R0)}; \hspace{0.5cm} \text{Mutex check} \\
\hspace{1cm} \text{BNE R6, R0, P} \hspace{0.5cm} ; \hspace{0.5cm} \text{If not 0, spin}

\hspace{1cm} \text{LW R3, head(R0)} \hspace{0.5cm} ; \hspace{0.5cm} \text{Load queue head into R3} \\
\text{spin: LW R4, tail(R0)} \hspace{0.5cm} ; \hspace{0.5cm} \text{Load queue tail into R4} \\
\hspace{1cm} \text{BEQ R4, R3, spin} \hspace{0.5cm} ; \hspace{0.5cm} \text{If queue empty,} \\
\hspace{1cm} \text{LW R5, 0(R3)} \hspace{0.5cm} ; \hspace{0.5cm} \text{Read x from queue into R5} \\
\hspace{1cm} \text{ADDI R3, R3, 4} \hspace{0.5cm} ; \hspace{0.5cm} \text{Shift head by one word} \\
\hspace{1cm} \text{SW R3 head(R0)} \hspace{0.5cm} ; \hspace{0.5cm} \text{Update head memory addr}

V: \hspace{1cm} \text{SW R0 mutex(R0)} \hspace{0.5cm} ; \hspace{0.5cm} \text{Give up mutex}

Assuming sequential consistency: 3 \text{MEMBAR}s not shown ...

What if the OS swaps a process out while in the critical section? "High-latency locks", a source of Linux audio problems (and others)
Non-blocking consumer synchronization

Another atomic read-modify-write instruction:

```c
Compare&Swap(Rt, Rs, m)
if (Rt == M[m])
    then
        M[m] = Rs; Rs = Rt; /* do swap */
else
    /* do not swap */
```

Assuming sequential consistency: MEMBARs not shown ...

```
try:    LW R3, head(R0) ; Load queue head into R3
spin:   LW R4, tail(R0) ; Load queue tail into R4
        BEQ R4, R3, spin ; If queue empty, wait
        LW R5, 0(R3)     ; Read x from queue into R5
        ADDI R6, R3, 4   ; Shift head by one word
        Compare&Swap R3, R6, head(R0); Try to update head
        BNE R3, R6, try  ; If not success, try again
```

If R3 != R6, another thread got here first, so we must try again.

If thread swaps out before Compare&Swap, no latency problem; this code only "holds" the lock for one instruction!
Semaphores with just \( \text{Lw} \) & \( \text{Sw} \)?

Can we implement semaphores with just normal load and stores? Yes!
Assuming sequential consistency ...

In practice, we create sequential consistency by using memory fence instructions ... so, not really “normal”.

Since load and store semaphore algorithms are quite tricky to get right, it is more convenient to use a \textbf{Test&Set} or \textbf{Compare&Swap} instead.
Conclusions: Synchronization

Memset: Memory fences, in lieu of full sequential consistency.

Test&Set: A spin-lock instruction for sharing write access.

Compare&Swap: A non-blocking alternative to share write access.
On Thursday

Part II ...

| Th 3/6 | Cache Design and Coherence | Chapter 3.12, Chapter 5.2-4. |

Have fun in section!