Parallelism and the Memory Hierarchy

Todd C. Mowry, Dave Eckhardt, Dave O’Hallaron & Brian Railing

I. Brief History of Parallel Processing

II. Impact of Parallelism on Memory Protection

III. Cache Coherence
A Brief History of Parallel Processing

• Initial Focus (starting in 1970’s): “Supercomputers” for Scientific Computing

C.mmp at CMU (1971)
16 PDP-11 processors

Cray XMP (circa 1984)
4 vector processors

Thinking Machines CM-2 (circa 1987)
65,536 1-bit processors + 2048 floating-point co-processors

SGI UV 1000cc-NUMA (today)
4096 processor cores

Blacklight at the Pittsburgh Supercomputer Center
A Brief History of Parallel Processing

- Initial Focus (starting in 1970’s): “Supercomputers” for Scientific Computing
- Another Driving Application (starting in early 90’s): Databases

Sun Enterprise 10000 (circa 1997)
- 64 UltraSPARC-II processors

Oracle SuperCluster M6-32 (today)
- 32 SPARC M2 processors
A Brief History of Parallel Processing

- Initial Focus (starting in 1970’s): “Supercomputers” for Scientific Computing
- Another Driving Application (starting in early 90’s): Databases
- Inflection point in 2004: Intel hits the Power Density Wall

Pat Gelsinger, ISSCC 2001
Impact of the Power Density Wall

• The real “Moore’s Law” continues
  – i.e. # of transistors per chip continues to increase exponentially

• But thermal limitations prevent us from scaling up clock rates
  – otherwise the chips would start to melt, given practical heat sink technology

• How can we deliver more performance to individual applications?
  → increasing numbers of cores per chip

• Caveat:
  – in order for a given application to run faster, it must exploit parallelism
Parallel Machines Today

Examples from Apple’s product line:

- **Mac Pro**: 12 Intel Xeon E5 cores
- **iPhone 6**: 2 A8 cores
- **iPad Air 2**: 3 A8X cores
- **iMac**: 4 Intel Core i5 cores
- **MacBook Pro Retina 15”**: 4 Intel Core i7 cores

(Images from apple.com)
Example “Multicore” Processor: Intel Core i7

- **Cores**: six 3.33 GHz Nahelem processors (with 2-way “Hyper-Threading”)
- **Caches**: 64KB L1 (private), 256KB L2 (private), 12MB L3 (shared)
Impact of Parallel Processing on the Kernel (vs. Other Layers)

- **Kernel itself becomes a parallel program**
  - avoid bottlenecks when accessing data structures
    - lock contention, communication, load balancing, etc.
    - use all of the standard parallel programming tricks
- **Thread scheduling gets more complicated**
  - parallel programmers usually assume:
    - all threads running *simultaneously*
      - load balancing, avoiding synchronization problems
    - threads *don’t move* between processors
      - for optimizing communication and cache locality
- **Primitives for naming, communicating, and coordinating need to be fast**
  - Shared Address Space: *virtual memory management* across threads
  - Message Passing: low-latency *send/receive* primitives

---

Mowry, Eckhardt, & O'Hallaron

15-410: Parallel Mem Hierarchies
Case Study: Protection

• One important role of the OS:
  – provide protection so that buggy processes don’t corrupt other processes

• Shared Address Space:
  – access permissions for virtual memory pages
    • e.g., set pages to read-only during copy-on-write optimization

• Message Passing:
  – ensure that target thread of send() message is a valid recipient
How Do We Propagate Changes to Access Permissions?

• What if parallel machines (and VM management) simply looked like this:
  – *(assume that this is a parallel program, deliberately sharing pages)*

  ![Diagram of parallel machines and memory hierarchy]

  **CPU 0**  **CPU 1**  **CPU 2**  …  **CPU N**

  **Page Table**

  **Memory**

  Set Read-Only

  - Read-Only  f029
  - Read-Only  f835

• Updates to the page tables (in shared memory) could be read by other threads
• But this would be a very slow machine!
  – Why?
"TLB Shootdown":

– relevant entries in the TLBs of other processor cores need to be flushed
1. Initiating core triggers OS to lock the corresponding Page Table Entry (PTE)
2. OS generates a list of cores that may be using this PTE (erringly conservatively)
3. Initiating core sends an Inter-Processor Interrupt (IPI) to those other cores
   – requesting that they invalidate their corresponding TLB entries
4. Initiating core invalidates local TLB entry; waits for acknowledgements
5. Other cores receive interrupts, execute interrupt handler which invalidates TLBs
   – send an acknowledgement back to the initiating core
6. Once initiating core receives all acknowledgements, it unlocks the PTE
**TLB Shootdown Timeline**

Performance of TLB Shootdown

- **Expensive operation**
  - e.g., over 10,000 cycles on 8 or more cores
- Gets more expensive with increasing numbers of cores

Now Let’s Consider Consistency Issues with the Caches
Caches in a Single-Processor Machine (Review from 213)

• Ideally, memory would be arbitrarily fast, large, and cheap
  – unfortunately, you can’t have all three (e.g., fast → small, large → slow)
  – cache hierarchies are a hybrid approach
    • if all goes well, they will behave as though they are both fast and large

• Cache hierarchies work due to locality
  – temporal locality → even relatively small caches may have high hit rates
  – spatial locality → move data in blocks (e.g., 64 bytes)

• Locating the data:
  – **Main memory**: directly (geographically) addressed
    • we know exactly where to look:
      – at the unique location corresponding to the address
  – **Cache**: may or may not be somewhere in the cache
    • need *tags* to identify the data
    • may need to check multiple locations, depending on the degree of associativity
Cache Read

$E = 2^e$ lines per set

$S = 2^s$ sets

$B = 2^b$ bytes per cache block (the data)

Address of word:
- $t$ bits
- $s$ bits
- $b$ bits

- Locate set
- Check if any line in set has matching tag
- Yes + line valid: hit
- Locate data starting at offset

valid bit
Intel Quad Core i7 Cache Hierarchy

Processor package

Core 0

- Regs
- L1 D-cache
- L1 I-cache
- L2 unified cache

- L3 unified cache (shared by all cores)

Core 3

- Regs
- L1 D-cache
- L1 I-cache
- L2 unified cache

L1 I-cache and D-cache: 32 KB, 8-way, Access: 4 cycles

L2 unified cache: 256 KB, 8-way, Access: 11 cycles

L3 unified cache: 8 MB, 16-way, Access: 30-40 cycles

Block size: 64 bytes for all caches.
Simple Multicore Example: Core 0 Loads X

load r1 ← X
(r1 = 17)

Main memory
X: 17
(4) Retrieve from memory, fill caches
Example Continued: Core 3 Also Does a Load of X

load \( r2 \leftarrow X \)

\( (r2 = 17) \)

Example of constructive sharing:
- Core 3 benefited from Core 0 bringing \( X \) into the L3 cache

Fairly straightforward with only loads. But what happens when stores occur?
What happens here?

We need to make sure we don’t have a consistency problem.

store 5 \rightarrow X
Options for Handling Stores in Uniprocessor Caches

Option 1: Write-Through Caches
→ propagate “immediately”
store 5 → X

What are the advantages and disadvantages of this approach?
Options for Handling Stores in Uniprocessor Caches

Option 1: Write-Through Caches
- propagate immediately
store 5 → X

Option 2: Write-Back Caches
- defer propagation until eviction
- keep track of dirty status in tags

(Analogous to PTE dirty bit.)

Upon eviction, if data is dirty, write it back.

Write-back is more commonly used in practice (due to bandwidth limitations)
Resuming Multicore Example

1. store 5 \(\rightarrow\) X

2. load r2 \(\leftarrow\) X \(r2 = 17\) Hmm...

What is supposed to happen in this case?

Is it incorrect behavior for r2 to equal 17?
• if not, when would it be?

(Note: core-to-core communication often takes tens of cycles.)
What is Correct Behavior for a Parallel Memory Hierarchy?

• Note: side-effects of writes are only observable when reads occur
  – so we will focus on the values returned by reads

• Intuitive answer:
  – reading a location should return the latest value written (by any thread)

• Hmm... what does “latest” mean exactly?
  – within a thread, it can be defined by program order
  – but what about across threads?
    • the most recent write in physical time?
      – hopefully not, because there is no way that the hardware can pull that off
        » e.g., if it takes >10 cycles to communicate between processors, there is
          no way that processor 0 can know what processor 1 did 2 clock ticks ago
    • most recent based upon something else?
      – Hmm...
Refining Our Intuition

What would be some clearly illegal combinations of \((A, B, C)\)?

How about:

\((4,8,1)? \quad (9,12,3)? \quad (7,19,31)?\)

What can we generalize from this?

- writes from any particular thread must be consistent with program order
  - in this example, observed even numbers must be increasing (ditto for odds)
- across threads: writes must be consistent with a valid interleaving of threads
  - not physical time! (programmer cannot rely upon that)
• Each thread proceeds in program order
• Memory accesses interleaved (one at a time) to a single-ported memory
  – rate of progress of each thread is unpredictable
Correctness Revisited

Recall: “reading a location should return the latest value written (by any thread)”

→ “latest” means consistent with some interleaving that matches this model
  – this is a hypothetical interleaving; the machine didn’t necessary do this!

Thread 0

// write evens to X
for (i=0; i<N; i+=2) {
    X = i;
    ...
}

Thread 1

// write odds to X
for (j=1; j<N; j+=2) {
    X = j;
    ...
}

Thread 2

... A = X;
... B = X;
... C = X;
...

CPU 0
CPU 1
CPU 2

Memory

Single port to memory
Two Parts to Memory Hierarchy Correctness

1. “Cache Coherence”
   – do all loads and stores to a **given cache block** behave correctly?
     • i.e. are they consistent with our interleaving intuition?
     • **important**: separate cache blocks have independent, unrelated interleavings!

2. “Memory Consistency Model”
   – do all loads and stores, even to **separate cache blocks**, behave correctly?
     • builds on top of cache coherence
     • especially important for synchronization, causality assumptions, etc.
Cache Coherence (Easy Case)

- One easy case: a physically shared cache

L3 cache is physically shared by on-chip cores
Cache Coherence: Beyond the Easy Case

• How do we implement L1 & L2 cache coherence between the cores?
  
  store 5 → X

• Common approaches: update or invalidate protocols
One Approach: **Update** Protocol

- **Basic idea:** upon a write, *propagate new value* to shared copies in peer caches

```plaintext
store 5 \rightarrow X
```

![Diagram showing the update protocol](image)

---

Carnegie Mellon

15-410: Parallel Mem Hierarchies

Mowry, Eckhardt, & O'Hallaron
Another Approach: **Invalidate** Protocol

- **Basic idea:** to perform a write, first delete any shared copies in peer caches

  \[ \text{store } 5 \rightarrow X \]
Update vs. Invalidate

• When is one approach better than the other?
  – *(hint: the answer depends upon program behavior)*

• Key question:
  – Is a block written by one processor read by others before it is rewritten?
    – if so, then *update* may win:
      • readers that already had copies will not suffer cache misses
    – if not, then *invalidate* wins:
      • avoids useless updates (including to dead copies)

• Which one is used in practice?
  – *invalidate* (due to hardware complexities of update)
    • although some machines have supported both (configurable per-page by the OS)
How Invalidation-Based Cache Coherence Works (Short Version)

• Cache tags contain additional coherence state ("MESI" example below):
  – Invalid:
    • nothing here (often as the result of receiving an invalidation message)
  – Shared (Clean):
    • matches the value in memory; other processors may have shared copies also
    • I can read this, but cannot write it until I get an exclusive/modified copy
  – Exclusive (Clean):
    • matches the value in memory; I have the only copy
    • I can read or write this (a write causes a transition to the Modified state)
  – Modified (aka Dirty):
    • has been modified, and does not match memory; I have the only copy
    • I can read or write this; I must supply the block if another processor wants to read

• The hardware keeps track of this automatically
  – using either broadcast (if interconnect is a bus) or a directory of sharers
Performance Impact of Invalidation-Based Cache Coherence

• Invalidations result in a new source of cache misses!

• Recall that uniprocessor cache misses can be categorized as:
  – (i) cold/compulsory misses, (ii) capacity misses, (iii) conflict misses

• Due to the sharing of data, parallel machines also have misses due to:
  – (iv) true sharing misses
    • e.g., Core A reads X → Core B writes X → Core A reads X again (cache miss)
      – nothing surprising here; this is true communication
  – (v) false sharing misses
    • e.g., Core A reads X → Core B writes Y → Core A reads X again (cache miss)
      – What???
      – where X and Y unfortunately fell within the same cache block
Beware of False Sharing!

- It can result in a **devastating ping-pong effect** with a very high miss rate
  - plus wasted communication bandwidth
- **Pay attention to data layout:**
  - the threads above appear to be working independently, but they are not

```c
while (TRUE) {
    X += ... 
    ... 
}
```

```c
while (TRUE) {
    Y += ... 
    ... 
}
```

**Invalidations, cache misses**
How False Sharing Might Occur in the OS

- Operating systems contain lots of **counters** (to count various types of events)
  - many of these counters are **frequently updated**, but **infrequently read**
- Simplest implementation: a **centralized counter**

```c
// Number of calls to get_tid
int gettid_ctr = 0;

int gettid(void) {
    atomic_add(&gettid_ctr, 1);
    return (running->tid);
}

int get_gettid_events(void) {
    return (gettid_ctr);
}
```

- Perfectly reasonable on a sequential machine.
- But it **performs very poorly** on a parallel machine. Why?
  - each update of `get_tid_ctr` invalidates it from other processor’s caches
“Improved” Implementation: An Array of Counters

- Each processor updates its own counter
- To read the overall count, sum up the counter array

```c
int gettid_ctr[NUM_CPUs] = {0};

int gettid(void) {
    // exact coherence not required
    gettid_ctr[running->CPU]++;
    return (running->tid);
}

int get_gettid_events(void) {
    int cpu, total = 0;
    for (cpu = 0; CPU < NUM_CPUs; cpu++)
        total += gettid_ctr[cpu];
    return (total);
}
```

- Eliminates lock contention, but may still perform very poorly. Why?
  → False sharing!

No longer need a lock around this.
Faster Implementation: Padded Array of Counters

• Put each private counter in its own cache block.
  – (any downsides to this?)

```
struct {
    int get_tid_ctr;
    int PADDING[INTS_PER_CACHE_BLOCK-1];
} ctr_array[NUM_CPUs];

int get_tid(void) {
    ctr_array[running->CPU].get_tid_ctr++;
    return (running->tid);
}

int get_tid_count(void) {
    int cpu, total = 0;
    for (cpu = 0; CPU < NUM_CPUs; cpu++)
        total += ctr_array[cpu].get_tid_ctr;
    return (total);
}
```

Even better: replace PADDING with other useful per-CPU counters.
Parallel Counter Implementation Summary

Centralized:

Simple Array:

Mutex contention? False sharing?

Padded Array:

Clustered Array:

Wasted space? (If there are multiple counters to pack together.)
• With **true sharing**, **spatial locality** can result in a **prefetching** benefit
• Hence **data layout can help or harm** sharing-related misses in parallel software
Summary

• Case study: memory protection on a parallel machine
  – TLB shootdown
    • involves Inter-Processor Interrupts to flush TLBs
• Part 1 of Memory Correctness: Cache Coherence
  – reading “latest” value does not correspond to physical time!
    • corresponds to latest in hypothetical interleaving of accesses
  – new sources of cache misses due to invalidations:
    • true sharing misses
    • false sharing misses

• Looking ahead: Part 2 of Memory Correctness, plus Scheduling Revisited