Shared Memory (Single Address Space) Basics

Nima Honarmand
Shared-Memory Programming Model
Programming Model

• Multiple execution contexts sharing a single address space
  – Multiple programs (MIMD)
  – Or more frequently: multiple copies of one program (SPMD)

• Implicit (automatic) communication via loads and stores
  – Runtime (OS and hardware) responsible for correct communication
Programming Model

• What about the four aspects of a parallel computing model?

1. Naming?
   – Pointers (memory addresses)

2. Communication?
   – Reading and writing shared memory locations

3. Synchronization
   – Conventional: Locks, barriers, conditional variables, ...
   – More recent: Transactional memory, ...

4. Work distribution?
   – Many forms: thread-parallel, task-parallel, ...

For now, we focus on the Thread-Parallel Model
Why Shared Memory?

• Pluses
  + Programmers don’t need to learn about explicit communications
    • Because communication is implicit (through memory)
  + Applications similar to the case of multitasking uniprocessor
    • Programmers already know about synchronization
  + OS needs only evolutionary extensions

• Minuses
  – Communication is hard to optimize
    • Because it is implicit
    • Not easy to get good performance out of shared-memory programs
  – Synchronization is complex
    • Over-synchronization $\rightarrow$ bad performance
    • Under-synchronization $\rightarrow$ incorrect programs
    • **Very** difficult to debug
  – Hard to implement in hardware

Result: the most popular form of parallel programming
Thread-Parallel Model

• One of many shared-memory programming models
  – Most primitive one (closest to hardware), hence the basis for others

• Each thread is a *logical processor*, executing a *sequence of statements*
  – Statement: high-level language statement or assembly instructions

• Each thread accesses two types of memory locations
  – *Private*: only read/written by that thread – *should conform to sequential semantics*
    • “Read A” should return the result of the last “Write A” (in thread order)
  – *Shared*: accessed by more than one thread – *what about these?*

• Answer is determined by the *Memory Consistency Model* of the system
Memory Consistency Model

• Or just *Memory Model*
  – Determines the answer to the previous question

• Determines the order in which shared-memory accesses from different threads can “appear” to execute
  – In other words, determines what value(s) a read can return
  – More precisely, *the set of all writes (from all threads) whose value can be returned by a read*

• (Should be) defined for any parallel system
  – Shared memory or message passing

• (Should be) defined at any layer of abstraction
  – High-level programming language or Hardware ISA

• For now, let’s focus on language-level memory models
• Compiler and hardware together responsible for guaranteeing language-level memory model
Intuitive Model: Sequential Consistency

```c
X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;
```

**sequential consistency (SC)**
[Lamport 1979]

Memory operations appear to occur in some **global** order consistent with the per-thread **program order**
Sequential Consistency

\[
\begin{align*}
X* \ x &= \text{null;} \\
\text{bool flag} &= \text{false;}
\end{align*}
\]

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;

In C++ model this can crash!
Intuitive reasoning fails in C++/Java

```c
X* x = null;
bool flag = false;
```

// Producer          // Consumer
A: x = new X();  C: while(!flag);
B: flag = true;  D: x->f++;

B doesn’t depend on A. It might be faster to reorder them!
Why are accesses reordered?

**Programming Language**
-sequentially valid optimizations can reorder memory accesses
  - e.g. common subexpression elimination, register promotion, instruction scheduling

**Compiler**
-sequentially valid optimizations can reorder memory accesses
  - e.g. out-of-order execution, store buffers

**Data-Race-Free-0 Model**
- Basis for Java and C++ Memory Model
A Short Detour: Memory Races

Two memory operations race if they
- are conflicting,
- are from different processors (threads), and
- There is a (sequentially consistent) execution in which are simultaneously ready to execute

// Thread t // Thread u
A: x = new Data(); C: while(!flag);
B: flag = true; D: x->f++;

Data Race
Useful Races

- Races are essential for implementing shared-memory synchronization

```c
AcquireLock()
{
    while (lock == 1) {} 
    t = CAS (lock, 0, 1); 
    if (!t) retry;
}
```

```c
ReleaseLock()
{
    lock = 0;
}
```
A program is **data-race-free** if all races are appropriately annotated (E.g., using `volatile` in Java / `atomic` in C++11)

Basically, saying that each memory access is either synchronization or data
- Synchronization accesses can race but should be annotated
- Data accesses should never race

**DRF0** [Adve & Hill 1990]

SC behavior for data-race-free programs, **weak** or no semantics otherwise

Java Memory Model (JMM) [Manson et al. 2005]

C++0x Memory Model [Boehm & Adve 2008]

Compiler/libraries should translate annotated instructions to special hardware instructions (atomic operations, memory fences, etc.) to ensure correct semantics.
DRF0-compliant Program

```cpp
X* x = null;
std::atomic<bool> flag = false;
```

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;  

• DRF0 guarantees SC
  .... only if data-race-free (all unsafe accesses are annotated)

• What if there is one data-race?
  .... C++ says all bets are off
  (e.g., compiler can output an empty binary!)
Shared-Memory System Architecture
Shared-Memory Physical Architecture

- Almost always there is a hierarchical structure
  - Within socket, within a machine, across machines
  - Contrary to the flat view of shared-memory programming models

- At each level of hierarchy, there are
  - multiple *processing elements* (cores, sockets, boxes, ...)
  - multiple *memory elements* (caches, DRAM modules, ...)
  - an *interconnection network* connecting them together

- At each level, one of two general configurations w.r.t. processor/memory connection

---

**Decoupled Configuration**

**Coupled Configuration**
Who Makes the Memory Shared?

- **Hardware shared memory**: hardware routes memory requests from processor to mem. element containing the requested location
  - And maintains “cache coherence” in the process

- **Software shared memory**: software makes remote memory accessible

- Example: *Distributed Shared Virtual Memory*
  - Threads see a single unified virtual memory space
  - Virtual memory manager (in the OS) distributes virtual memory pages across multiple machines
  - Uses network to deliver a page to the requesting machine

- HW SM makes physical memory shared
- SW SM makes a higher abstraction (e.g., virt mem) shared

In CSE610, we’ll focus on HW shared memory.
Shared vs. Point-to-Point Networks

• Shared medium (a.k.a. bus)
  – All elements connected to a shared physical medium
  – Low latency, low bandwidth
  – Doesn’t scale to large number of elements
  – Simpler protocols (e.g., cache coherence)

• Point-to-point networks
  – Each link is only connected to two end points
  – Many examples: fully connected, ring, crossbar, (fat) tree, mesh, torus, hypercube, ...
  – High latency (many “hops”), higher bandwidth per element
    • Scales to 1000s of elements
  – Complex protocols (e.g., cache coherence)
Example: Chip-level Interconnect

- Intel Xeon® E5-2600
  - Double ring interconnect
  - Connecting 8 cores and 8 banks of L3

Example: Node-Level Interconnect

- Intel Quick Path Interconnect (QPI)
  - Fully connected
  - Connecting processor sockets to each other and IO hubs
  - Memory directly connected to processor sockets using a memory bus

Example: Node-Level Interconnect

• Sun Starfire Interconnect
  – Separate Address and Data networks
  – Partitioned bus for address
    • Bus to simplify coherence protocol
    • Partitioned to improve bandwidth
  – Crossbar for data (to improve bandwidth)

Source: http://www.filibeto.org/~aduritz/truetrue/e10000/starfire-interconnect.html
Example: Rack-Level Interconnect

- **SGI Altix 4700**
  - Crossbar switches
  - Connecting nodes (server blades) to each other

Source: http://techpubs.sgi.com/library/manuals/4000/007-4823-001/sgi_html/ch03.html
Organizing Point-to-Point Networks

- Network topology: organization of network
  - Tradeoff performance (connectivity, latency, bandwidth) and cost

- Router chips
  - Networks that require separate router chips are *indirect*
  - Networks that use processor/memory/router packages are *direct*
    - Fewer components, “Glueless MP”
Uniformity in Memory Access

• **Uniform Memory Access (UMA):** Equal latency to memory from all processors
  – Simpler software, doesn’t matter where you put data
  – Lower peak performance
  – Common in bus-based systems

• **Non-Uniform Memory Access (NUMA):** Local memory access faster than remote
  – More complex software: where you put data matters
  – Higher peak performance: assuming proper data placement
Issues for Shared Memory Systems

• Two big ones
  – *Cache coherence*
  – *Memory consistency model*

• Closely related
  – *But often confused*

• A (slightly) smaller one (very important nonetheless)
  – *Synchronization support*
    • Special instructions, e.g., atomic operations such as x86 `compxchg`
    • Advanced features such as full/empty bits