Announcements

- Homeworks
  - still awaiting HW1 (1), HW2 (2), HW3 (3), HW4 (12)

- Project
  - Part 1 of the writeup due today, Part 2 due 12/14
  - I am available if you want to discuss approach/problems

- Lectures
  - No lecture next week (Thanksgiving)
  - 12/3: last lecture
  - 12/10: project presentations (7 groups)
    - problem (5 minutes), approach (5 minutes), results (5 minutes)

- Events of interest
  - November 20th: PARCON’98, Main Building Room 703
  - December 4th: Jaswinder Pal Singh (Princeton)

Outline

- Last lecture: Programming for Performance (contd.)
  - message-passing programming models
    - components of communication cost
      - synchronization (matching), overhead, latency, BW, contention
    - synchronization: pre-posted receives, message pipelining, multithreading
    - overhead: message aggregation, low-overhead messaging layers
    - contention: scheduling communication

- This lecture:
  - programming for performance (contd.)
    - shared-memory models
      - memory access costs, synchronization overheads
    - hardware/software tradeoffs
      - shared virtual memory
        - software access control models
        - relaxed memory consistency

[ Culler/Singh/Gupta: Chapter 9, Section 9.3 ]

Performance of Shared Memory Programs

- Division of responsibility between hardware and programmer
  - hardware provides efficient data access and coherence primitives
  - programmer writes program to take advantage of these primitives

- Performance determined by two factors
  - memory access costs
    - does the program utilize the cache structure effectively?
  - temporal locality: small working sets so they fit in the cache
    - blocked algorithms as in the Equation Solver, LU examples
  - spatial locality: reduce fragmentation and false sharing
  - synchronization costs
    - how much time do threads spend waiting
      - for another thread (waiting for an event)
      - for entry into a critical section
Recap: Improving Spatial Locality

- Reduce spatial interleaving of accesses
  - (task assignment) contiguous assignment of array elements
    - avoid ping-ponging of cache blocks for writes
  - (data structuring) higher-dimensional arrays to keep partitions contiguous

Recap: Improving Spatial Locality (contd.)

- Use per-processor heaps for dynamic memory allocation
  - ensures data structures use different cache blocks

- Copy data to increase locality
  - e.g., reuse of noncontiguous data
  - must trade off against cost of copying

- Pad and align arrays
  - can have false sharing versus fragmentation tradeoff

- Organize arrays of records for spatial locality
  - e.g. particles with fields: organize by particle or by field
    - in vector programs by field for unit-stride, in parallel often by particle
  - phases of program may have different access patterns and needs

Reducing Synchronization Costs

Two degrees of freedom for reducing synchronization costs

- Improve performance of lock primitives
  - acquire method: wait to go past synchronization point
    - contention
    - deadlock, livelock, and progress
    - blocking versus non-blocking (lock-free) implementations
  - waiting method: what do processes do when waiting for a lock
    - busy-waiting (spinning) versus blocking
    - OS preemption and scheduling concerns

- Restructure program to reduce synchronization overheads
  - reduce the size of critical sections
  - partition/fuse critical sections
  - implicit synchronization (data ownership)
  - reader-writer locks
Better Lock Primitives: Ticket Lock

- Works like waiting line at deli or bank
  - two counters per lock (next_ticket, now_serving)
  - acquire: fetch&inc next_ticket; wait for now_serving to equal it
  - atomic op when arrive at lock, not when it’s free (so less contention)
  - release: increment now-serving

Benefits
- FIFO order
- low latency for low-contention if fetch&inc cacheable

Costs
- O(p) read misses at release, since all spin on same variable
- can be difficult to find a good amount to delay on backoff
  - exponential backoff not a good idea due to FIFO order
  - backoff proportional to now-serving - next-ticket may work well

Wouldn’t it be nice to poll different locations ...

Better Lock Primitives: Array-based Queuing Locks

- Waiting processes poll on different locations in an array of size p
  - acquire
    - fetch&inc to obtain address on which to spin (next array element)
    - ensure that these addresses are in different cache lines or memories
  - release
    - set next location in array, thus waking up process spinning on it

Benefits
- O(1) traffic per acquire with coherent caches
- FIFO ordering, as in ticket lock

Costs
- O(p) space per lock

Preemption and Scheduling-Conscious Locks

- Problem
  - what should happen if process holding the lock gets preempted?
  - … process is stuck in the scheduler queue with low priority?

- Active area of research: Michael Scott (U. Rochester)

- Some strategies
  - avoidance: request that the kernel not preempt a thread holding the lock
  - warning: kernel warns process some time prior to preemption
    - if process has not been warned when it tries to enter the critical section, it can complete it without preemption, else it must yield
  - recovery: process waiting to enter a critical section can “donate” its scheduling quanta to the process that is holding the lock
  - all of these must also factor in kernel process scheduling

Non-blocking Lock Primitives

- Built on top of non-blocking universal atomic primitives
  - LL/SC: SC succeeds only if no intervening SC
  - CAS: compare-and-swap

- Overall strategy
  - if a process falls behind, some other process completes the operation for it
  - provides automatic tolerance to preemption- and scheduling behavior
  - however, primitives are data-structure specific
    - e.g., concurrent queues, heaps, etc.
  - [discuss code in handout]
Program Restructuring to Reduce Lock Overheads

- **Reduce size of critical sections**
  - only those data structures that are subject to concurrent access
  - *advantage*: reduces contention, waiting time
- **Partition critical sections**
  - independent portions of the program get their own critical sections
  - *advantage*: no false serialization
- **Fuse critical sections**
  - tradeoff between synchronization overhead and amount of concurrency
- **Implicit synchronization (data ownership)**
  - eliminate small critical sections by relying on fact that only one process would be updating it
- **Reader-writer locks**
  - allow more concurrency by exploiting operation semantics
  - e.g., hash-table update versus hash-table lookup

Programming for Performance: Summary

- Programming models represent different division of responsibility between hardware, system software, and applications (programmer)
  - data parallel models
    - programmer specifies data distribution (load-balance, locality)
    - compiler support reduces programming effort
  - message-passing models
    - programmer specifies everything!
    - but, minimal hardware and system software costs
  - shared memory models
    - main concerns: locality and synchronization costs
    - high degree of hardware support reduces programming effort
- Remainder of lecture: Can we reduce hardware costs of supporting the shared memory model?
  - advantages: reduced programming effort + reduced cost

Hardware Costs of CC-NUMA Machines

Four main components of the communication architecture
- checking for *access violation*
- per-block tags and state (directory)
- *protocol processing*
  - including interventions in the processor cache
- network interface
- Ideally, these four components are tightly integrated
  - few bus crossings, dedicated data and control paths
- Active area of research: Reducing amount of hardware support
  - for detecting access violations
  - for protocol processing
  - for loosely-coupled network interfaces (communication is expensive)

Hardware Access Control w/ Decoupled Assist

- Add (small) special hardware to *snoop* on loads/stores issuing from the processor and then take appropriate action
  - protocol processing can be done in software, or on a separate processor
  - tags can be kept in main-memory or in special memory
  - assume that the network interface has support for fine-grained messages

<table>
<thead>
<tr>
<th>SGI Origin 2000</th>
<th>Wisconsin Typhoon (moderate decoupling)</th>
<th>Wisconsin Typhoon (complete decoupling)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>$</td>
<td>CPU</td>
</tr>
<tr>
<td>AC+NI+PP</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td>Mem</td>
<td>AC+NI+PP</td>
<td>PP</td>
</tr>
<tr>
<td>CPU</td>
<td>$</td>
<td>CPU</td>
</tr>
<tr>
<td>AC+NI+PP</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td>Mem</td>
<td>AC</td>
<td>NI</td>
</tr>
</tbody>
</table>
Access Control through Code Instrumentation

- *Idea:* Instead of hardware checks for access violation, have the compiler insert code before each load/store instruction which verifies that the memory location has the desired access rights.

```c
line += 3;
v = v - line;
```

Compile

```c
push ptr[line]
call __check_w
load r1, ptr[line]
push ptr[v]
call __check_w
load r2, ptr[v]
add r1, 3h
store r1, ptr[line]
push ptr[line]
call __check_r
sub r2, r1
push ptr[line]
call __done
store r2, ptr[v]
push ptr[v]
call __done
```

Opt.

```c
push ptr[line]
call __check_w
load r1, ptr[line]
push ptr[v]
call __check_w
load r2, ptr[v]
add r1, 3h
store r1, ptr[line]
push ptr[line]
call __check_r
sub r2, r1
push ptr[line]
call __done
store r2, ptr[v]
push ptr[v]
call __done
```

Page-Based Access Control

- *Idea:* Leverage virtual-memory support provided by the memory management units of microprocessors and by the OS.
- Simplest proposal: Ivy (Li and Hudak, 1989)
  - sharing granularity: a virtual memory page
  - access violations based on mapping (invalid, read-only, read/write)
  - protocol processing in page-fault handlers

Shared virtual address space

SVM Library

Local memories of nodes

Access Control Using Language/Compiler Support

- *Idea:* Have the programmer deal with data “regions” (objects) instead of memory, and allow language/compiler to specify when a region is being accessed and in which mode.

- Advantages
  - better match of sharing granularity to program structures
  - natural fit for high-level shared object languages (e.g., Java)
    - method entry and exit demarcate scope of access
  - natural use of even more relaxed consistency models: *Entry Consistency*
    - weaker than release consistency
    - distinguishes among different synchronization variables
      - all LD/ST operations within a synchronization block can be reordered
      - unlike RC, need to wait only for acquire/release operations to the same synchronization variable

Software Coherence Processing

- Invoked upon detecting an access violation
  - block tags and directory state maintained in software
  - explicit issuing/synchronization on network transactions (messages) sent to other nodes
  - explicit memory management: storage for cached blocks

- Challenge: How to overcome higher processing overheads?
- Advantage: Flexibility
  - everything is in software, so can do more complicated things
  - relaxed memory-consistency models
  - different protocols for different sharing patterns
    - read-only, producer-consumer, write-mostly, migratory sharing, etc.
  - a very active area of research
Problems with Basic Page-Based Shared Memory

- Large sharing granularity implies more fragmentation + false sharing
  - ping-pong of pages between nodes
  - expensive because it results in
    - more page-faults: interrupt + multiple kernel-user space crossings
    - substantial communication overhead to maintain SC

- Solution: Use more sophisticated protocols

Using Relaxed Memory Consistency

- Reduce required amount of communication
  - release consistency (RC) allows invalidations to not be propagated until synchronization points
    - different flavor from use of RC in hardware for reducing write stalls
  - however, propagating invalidations on release still results in more communication than is strictly required
    - a process may not do an acquire (so it does not need to see the write notices)
    - separate messages are required for invalidations and acquires
    - may invalidate data earlier than necessary, causing additional false sharing

Relaxed Memory Consistency Models (contd.)

Eager release consistency: Propagate write-notices on release

Lazy release consistency: Propagate write-notices on acquire

- more complicated to implement (causality); details in Section 9.6.2

Multiple Writer Protocols

- LRC solves problems of false sharing but invalidations are still required if there are multiple writers
- Solution (used in TreadMarks)
  - initially write-protect a page
  - on first write, a protection violation occurs
    - system makes copy of the page (called a twin)
    - unprotect the page to allow further writes
  - associate a diff (page - twin) with the write-notice
Alternative Methods for Propagating Writes

- Problem with earlier solution (particularly with LRC):
  - processing overheads: diff creation
  - memory overhead: diffs and write-notices have arbitrary lifetime
    - since not possible to predict when a processor would do an acquire
  - need distributed garbage collection

- Recent solutions focus on home-based protocols
  - one node maintains “master” copy of node
  - diffs are eagerly folded into this copy
  - on an acquire
    - requesting processor gets write-notices from releasing processor
    - fetches whole page from home node
  - only one round-trip required to update page, instead of multiple messages

- diff costs can be further reduced by relying on some hardware support
  - writes are automatically propagated to the home node
    - Princeton SHRIMP, Digital Memory Channel

Page-Based Shared Memory: Implications

- LRC + multiple writer protocols improve performance dramatically
- However, still a wide gap as compared to hardware shared memory for applications with fine-grained sharing
  - false sharing
  - extra communication and processing overhead
  - page faults and fetches are expensive to satisfy
  - synchronization through software messages: dilates critical sections
  - scalability problems because of auxiliary data structures

- Research challenge
  - can one ever match performance of hardware shared memory using software-only approaches?
  - Key: software approaches can take advantage of protocol flexibility
    - customize protocols to application behavior
    - more processing but reduced communication (favored by architectural trends)

Lecture Summary

- Programming for performance (contd.)
  - shared memory models
    - components of cost: memory access, synchronization
    - reducing memory access cost
      - temporal and spatial locality
    - reducing synchronization cost
      - smaller critical sections, algorithms with reduced serialization

- Hardware/Software tradeoffs
  - shared virtual memory
    - access control: hardware, code instrumentation, page-based, language support
    - relaxed memory consistency
      - lazy release consistency
    - multiple writer protocols
    - implications for software

Next Lecture

- Future Directions
  - hardware and software trends
  - high-level programming models and compilation issues
  - parallel programming tools
  - concluding remarks