Announcements

- Homeworks and project writeups
  - past due
    - HW1 (1), HW2 (1), HW3 (3), HW4 (9)
    - part 1 of the writeup (15)
  - part 2 of the project writeup
  - hard deadline for everything: December 14th, 1998

- Project presentations: December 10th, 1998
  - 7 groups ➞ 15 minutes per group
    - problem (5 minutes), approach (5 minutes), results (5 minutes)
    - will announce order of presentations on mailing list (arbitrary)
      - you are expected to be present for the entire lecture

- Events of interest
  - December 4th: Jaswinder Pal Singh (Princeton)

Outline

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

- This lecture: Future directions
  - hardware/software tradeoffs (contd.)
    - relaxed memory consistency models
  - future directions and challenges
    - hardware and software trends
    - high-level programming models and compilation issues
    - parallel programming tools
  - closing remarks

Hardware/Software Tradeoffs: 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

- Let us consider example of page-based coherence protocols ...
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

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 auxillary data structures

彩神Research challenge彩神
  – 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)

Outline for Remainder of the Lecture

• Course summary

• Future directions
  – hardware and software trends
  – programming models and compilation issues
  – tools for parallel programming

• Research projects @NYU
  – system support for parallel and distributed computing
  – applications of high-performance computing

Course Summary

• Architecture and programming of parallel machines

Lecture 1: Introduction
  technology trends driving parallel computing
Lectures 2-3: Parallel programs and programming models
  data parallel, message passing, shared memory
Lectures 4-7: Parallel architectures
  small scale shared-memory
  large-scale distributed memory
  large-scale shared memory
Lectures 8-10: Programming for performance
  understanding mapping between programming models and architectures
Lecture 11: Future directions
Future Hardware Trends

The good news:
- Multiprocessing on the desktop
  - growing transistor budgets enable multiple processors on a single chip
  - integration with large amounts of memory
  ➔ pervasive use of parallel computing
  • your workstation will be a parallel computer

The bad news:
- Growing gap between processor and “memory”
  - communication is significantly more expensive than computation
  ➔ need for better orchestration and locality management (caches)

What might save the day:
- Trading off computation for communication
- Taking advantage of reconfigurable or “active” devices

Future Software Trends

- Component-based approaches
  - large-scale programs typically built from pre-existing pieces
  - implications: how best to reuse sequential components?
    • *shared-address space* approaches: permit reuse at cost of performance
    • *message passing* approaches: code might require major modification, but easier to get predictable performance
      • is there a middle ground?
- Parallel programs will have to run in more general environments
  - future large-scale parallel computers
    • built using complete workstation building blocks (economy of scale)
    • heterogeneous (by choice or because blocks span several generations)
  - implications
    • need for *functionality and performance*, portability
    • ability to tolerate faults, trust and security concerns
    • additional level of complexity for orchestration, locality management
      ➔ convergence of concerns for parallel and distributed computing

Trends in Programming Models

- Need for a programming model that is both “simple to use” and yet capable of delivering “high performance”
  - essential for making parallel programming commonplace
- Low-level approaches (currently, the most popular)
  - message-passing, shared-address space
    • debatable whether these are “simple to use”
- Very high-level approaches
  - data-parallel approaches: good for restricted classes of computation
  - declarative languages: functional, logic, and dataflow
    • may remain impractical for the foreseeable future
- More pragmatic approaches
  - compromise between ease of use and ability to achieve performance
  - one such approach is *concurrent object-oriented programming*

Concurrent Object-Oriented Languages (COOP)

- Extends shared-memory programming models
  - *global shared objects*: simplifies synchronization and concurrency control
  - *implicit thread creation*: enables natural expression of program structure
  - examples: Java, Compositional C++, Illinois Concert C++

- Advantages
  - encapsulation and modularity benefits of object-orientation
  - objects provide a natural granularity for locality management
COOP Models: Efficiency Challenges

- Portable and efficient support of high-level abstractions
  - global object space
  - fast communication mechanisms for remote method invocation
  - efficient object-level caching mechanisms
- implicit thread creation
  - support for fine-grained threads (much smaller-grained than pthreads)
  - low-overhead load-balancing mechanisms

Parallel Programming Tools

- Critical, but often overlooked component
- Parallel debuggers
  - more involved than just providing a dbx window per process
  - particularly with larger number of processors, and non-SPMD models
  - main challenge: eliminating race conditions (for deterministic playback)
  - active research area: how to obtain a global snapshot of computation state
- Performance monitors
  - message passing programs: ParaGraph, Pablo (UIUC)
  - timestamp message sends and receives and display graphically
  - shared-address space programs: relatively few tools
  - architectural support can help: e.g., per-data structure logs of cache behavior
- Computation steering tools
  - active research area: more active control of what gets monitored, and
    modification of computation on-the-fly
  - e.g., Falcon toolkit (Georgia Tech)

Research Projects @NYU

- History: The NYU Ultracomputer
- Current projects
  - system support for parallel and distributed computing
    - Allan Gottlieb: InterMemory
    - Vijay Karamcheti
      - Application-driven memory-hierarchy management
      - Computing Communities (with Zvi Kedem and Partha Dasgupta)
      - my research spans applications, compilers, run-time systems, OS, and architecture
    - Zvi Kedem: MILAN, Computing Communities
    - Krishna Palem: React-ILP
    - Dennis Shasha: P-LINDA
  - applications of high-performance computing
    - Scientific/Numerical Computing: Marsha Berger + several Math faculty
    - Databases: Dennis Shasha
    - Graphics/Visualization: Ken Perlin, Denis Zorin
    - Biology/Medicine: Bud Mishra

Application-Driven Memory Hierarchy Management

- Problem: Fine-grained software object shared memory on clusters
  - motivation: application migration path from small-scale SMPs
  - plus, convenient support of high-level programming models
  - key challenge: communication performance
    - 50μs versus 2μs on custom parallel machines

new sharing abstractions
+ analysis/optimization of application sharing behavior
+ efficient implementation
  (using "active" system components)
New Sharing Abstractions: View Caching

- Caching at the granularity of “views” enables customization
  - *what* portion of object state gets transferred
  - *when* coherence is required
  - *how* coherence is implemented

Computing Communities

- Enabling system infrastructure to allow
  - **dynamic aggregation** of computing and information resources (processors, networks, disks, databases) into “computing communities”
  - use of these “communities” for predictable execution of applications, where both application functionality and output quality are automatically adjusted based upon characteristics of available resources

Closing Remarks

- Parallel computing (definition due to Almasi/Gottlieb):
  “A collection of processing elements that can communicate and cooperate to solve large problems fast”

- “Communicate and cooperate”
  - node and interconnect architecture (Lectures 4, 5, 6, 7)
  - problem partitioning and orchestration (Lectures 2, 3)
- “Large problems fast”
  - programming model (Lectures 1, 2)
  - match of model/architecture (Lectures 8, 9, 10, 11)
- Outcomes of the course: An understanding of
  - dominant parallel programming models
  - dominant parallel architectures
  - interaction between models and architectures