CS4230 Parallel Programming

Lecture 3: Introduction to Parallel Architectures

Mary Hall
August 28, 2012

Homework 1: Parallel Programming Basics

Due before class, Thursday, August 30

Turn in electronically on the CADE machines using the handin program: "handin cs4230 hw1 <probfile>"

- Problem 1: (from today's lecture) We can develop a model for the performance behavior from the versions of parallel sum in today's lecture based on sequential execution time S, number of threads T, parallelization overhead O (fixed for all versions), and the cost B for the barrier or M for each invocation of the mutex. Let N be the number of elements in the list. For version 5, there is some additional work for thread 0 that you should also model using the variables above. (a) Using these variables, what is the execution time of valid parallel versions 2, 3 and 5; (b) present a model of when parallelization is profitable for version 3; (c) discuss how varying T and N impact the relative profitability of versions 3 and 5.

Homework 1: Parallel Programming Basics

- Problem 2: (#1.3 in textbook): Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, …).

  Hints: Use a variable `divisor` to determine whether a core should send its sum or receive and add. The `divisor` should start with the value 2 and be doubled after each iteration. Also use a variable `core_difference` to determine which core should be partnered with the current core. It should start with the value 1 and also be doubled after each iteration. For example, in the first iteration `0 % divisor = 0` and `1 % divisor = 1`, so 0 receives and adds, while 1 sends. Also in the first iteration `0 + core_difference = 1` and `1 - core_difference = 0`, so 0 and 1 are paired in the first iteration.

Today's Lecture

- Flynn's Taxonomy
- Some types of parallel architectures
  - Shared memory
  - Distributed memory
- These platforms are things you will probably use
  - CADE Lab1 machines (Intel Nehalem i7)
  - Sun Ultrasparc T2 (water, next assignment)
  - Nvidia GTX260 GPUs in Lab1 machines
- Sources for this lecture:
  - Textbook
  - Jim Demmel, UC Berkeley
  - Notes on various architectures

Reading this week: Chapter 2.1-2.3 in textbook

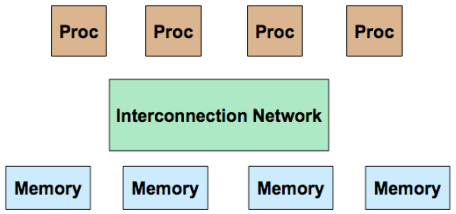
Chapter 2: Parallel Hardware and Parallel Software

2.1 Some background
- **The von Neumann architecture**
- Processes, multitasking, and threads

2.2 Modifications to the von Neumann Model
- **The basics of caching**
- Cache Mappings
- Caches and programs: an example
- Virtual memory
- **Instruction-level parallelism**
- **Hardware multithreading**

2.3 Parallel Hardware
- **SIMD systems**
- **MIMD systems**
- **Interconnection networks**
- **Cache coherence**
- **Shared-memory versus distributed-memory**

An Abstract Parallel Architecture



- How is parallelism managed?
- Where is the memory physically located?
- Is it connected directly to processors?
- What is the connectivity of the network?

Why are we looking at a bunch of architectures

- There is no canonical parallel computer – a diversity of parallel architectures
  - Hence, there is no canonical parallel programming language
- Architecture has an enormous impact on performance
  - And we wouldn't write parallel code if we didn't care about performance
- Many parallel architectures fail to succeed commercially
  - Can't always tell what is going to be around in N years

Challenge is to write parallel code that abstracts away architectural features, focuses on their commonality, and is therefore easily ported from one platform to another.

The von Neumann Architecture

Conceptually, a von Neumann architecture executes one instruction at a time

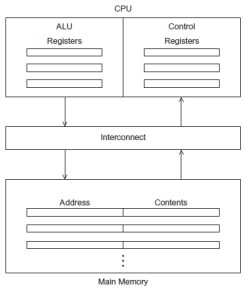
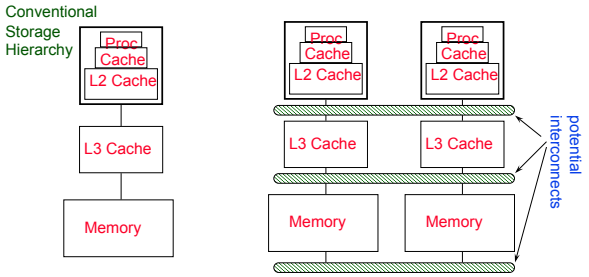


Figure 2.1

Locality and Parallelism



- Large memories are slow, fast memories are small
- Program should do most work on local data

Uniprocessor and Parallel Architectures

Achieve performance by addressing the von Neumann bottleneck

- Reduce memory latency
  - Access data from "nearby" storage: registers, caches, scratchpad memory
  - We'll look at this in detail in a few weeks
- Hide or Tolerate memory latency
  - Multithreading and, when necessary, context switches while memory is being serviced
  - Prefetching, predication, speculation
- Uniprocessors that execute multiple instructions in parallel
  - Pipelining
  - Multiple issue
  - SIMD multimedia extensions

How Does a Parallel Architecture Improve on this Further?

- Computation and data partitioning focus a single processor on a subset of data that can fit in nearby storage
- Can achieve performance gains with simpler processors
  - Even if individual processor performance is reduced, ***throughput*** can be increased
- Complements instruction-level parallelism techniques
  - Multiple threads operate on distinct data
  - Exploit ILP within a thread

Flynn's Taxonomy

classic von Neumann SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream not covered	(MIMD) Multiple instruction stream Multiple data stream

More Concrete: Parallel Control Mechanisms

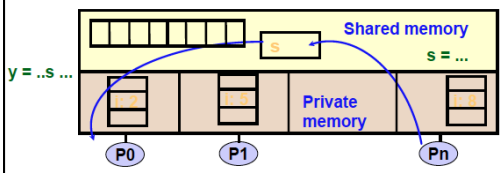
Name	Meaning	Examples
Single Instruction, Multiple Data (SIMD)	A single thread of control, same computation applied across "vector" elts	Array notation as in Fortran 90: `A[1:n] = A[1:n] + B[1:n]`
Multiple Instruction, Multiple Data (MIMD)	Multiple threads of control, processors periodically synch	Parallel loop: `forall (i=0; i<n; i++)`
Single Program, Multiple Data (SPMD)	Multiple threads of control, but each processor executes same code	Processor-specific code: `if ($myid == 0) { }`

Two main classes of parallel architecture organizations

- Shared memory multiprocessor architectures
  - A collection of autonomous processors connected to a memory system.
  - Supports a global address space where each processor can access each memory location.
- Distributed memory architectures
  - A collection of autonomous systems connected by an interconnect.
  - Each system has its own distinct address space, and processors must explicitly communicate to share data.
  - Clusters of PCs connected by commodity interconnect is the most common example.

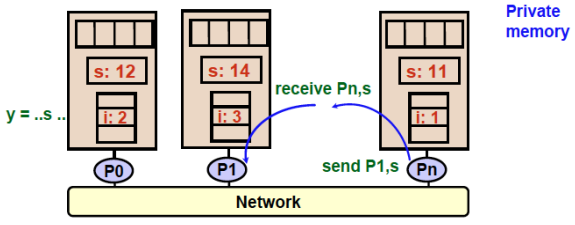
Programming Shared Memory Architectures

A shared-memory program is a collection of threads of control.
- Threads are created at program start or possibly dynamically
- Each thread has **private variables, e.g., local stack** variables
- Also a set of **shared variables,** e.g., static variables, shared common blocks, or global heap.
- Threads communicate **implicitly** by writing and reading **shared** variables.
- Threads coordinate through **locks** and **barriers** implemented using shared variables.

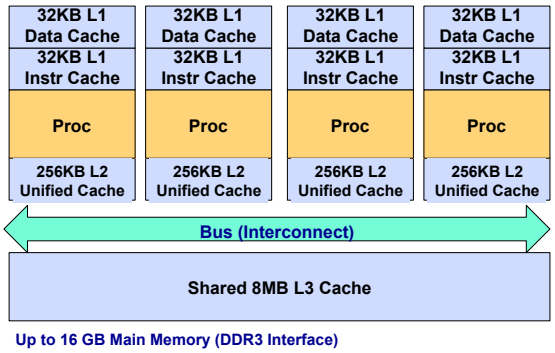


Programming Distributed Memory Architectures

A distributed-memory program consists of named processes.
- Process is a thread of control plus local address space -- NO shared data.
- Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
- Coordination is implicit in every communication event.



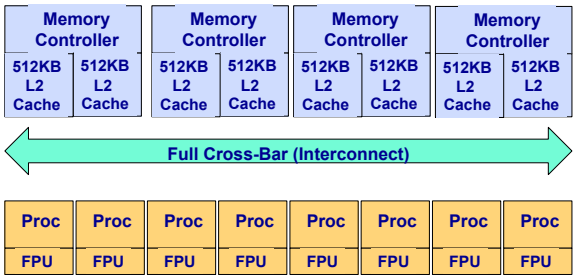
Shared Memory Architecture 1: Intel i7 860 Nehalem (CADE LAB1)



More on Nehalem and Lab1 machines -- ILP

- Target users are general-purpose
  - Personal use
  - Games
  - High-end PCs in clusters
- Support for SSE 4.2 SIMD instruction set
- 8-way hyperthreading (executes two threads per core)
- multiscalar execution (4-way issue per thread)
- out-of-order execution
- usual branch prediction, etc.

Shared Memory Architecture 2: Sun Ultrasparc T2 Niagara (water)



More on Niagara

- Target applications are server-class, business operations
- Characterization:
  - Floating point?
  - Array-based computation?
- Support for VIS 2.0 SIMD instruction set
- 64-way multithreading (8-way per processor, 8 processors)

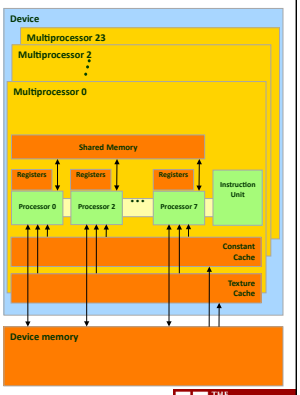
Shared Memory Architecture 3: GPUs
Lab1 has Nvidia GTX 260 accelerators

24 Multiprocessors, with 8 SIMD processors per multiprocessor

- SIMD Execution of warpsize threads (from single block)
- Multithreaded Execution across different instruction streams

Complex and largely programmer-controlled memory hierarchy

- Shared Device memory
- Per-multiprocessor "Shared memory"
- Some other constrained memories (constant and texture memories/caches)
- No standard data cache

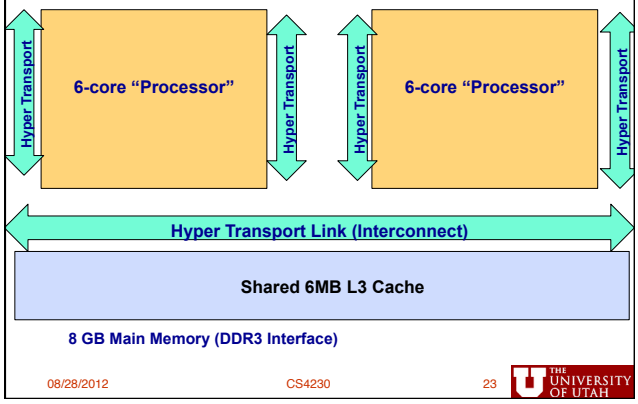


Jaguar Supercomputer

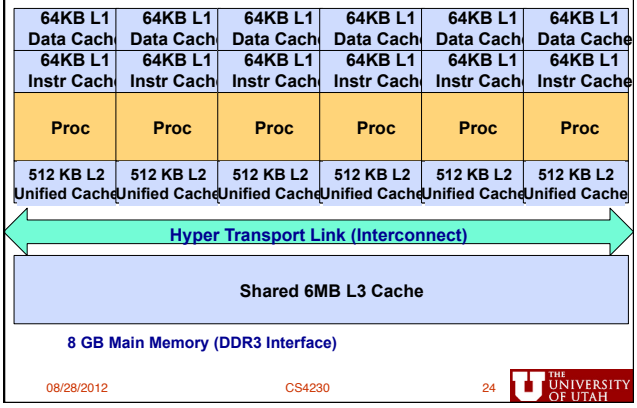
Peak performance of 2.33 Petaflops
224,256 AMD Opteron cores

http://www.olcf.ornl.gov/computing-resources/jaguar/

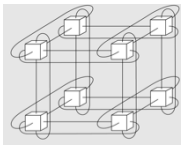
Shared Memory Architecture 4:
Each Socket is a 12-core AMD Opteron Istanbul



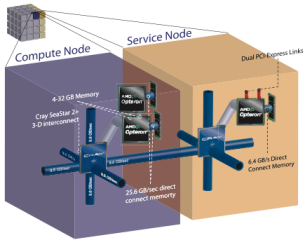
Shared Memory Architecture 3:
Each Socket is a 12-core AMD Opteron Istanbul



Jaguar is a Cray XT5 (plus XT4) Interconnect is a 3-d mesh



3-dimensional toroidal mesh



http://www.cray.com/Assets/PDF/products/xt/CrayXT5Brochure.pdf

Summary of Architectures

Two main classes

- Complete connection: CMPs, SMPs, X-bar
  - Preserve single memory image
  - Complete connection limits scaling to small number of processors (say, 32 or 256 with heroic network)
  - Available to everyone (multi-core)
- Sparse connection: Clusters, Supercomputers, Networked computers used for parallelism
  - Separate memory images
  - Can grow "arbitrarily" large
  - Available to everyone with LOTS of air conditioning
- Programming differences are significant

Brief Discussion

- Why is it good to have different parallel architectures?
  - Some may be better suited for specific application domains
  - Some may be better suited for a particular community
  - Cost
  - Explore new ideas
- And different programming models/ languages?
  - Relate to architectural features
  - Application domains, user community, cost, exploring new ideas