CSE 392/CS 378: High-performance Computing - Principles and Practice

Parallel Computer Architectures
A Conceptual Introduction for Software Developers

Jim Browne
browne@cs.utexas.edu
Parallel Computer Architectures
A Conceptual Introduction

Lecture Outline
• Why Do Applications Care about Architecture?
• Example Architecture – Ranger
• What Are the Issues?
• Why Do Application Developers/Users Need to Know about Parallel Architectures?
  – To efficiently develop efficient applications
• What Do Application Developers/Users Need to Know about Parallel Architectures?
  – Characteristics of Memory Hierarchy
  – Parallelism in Cluster Computer Architectures
    • Homogeneous Multicore Chip Architecture
    • Homogeneous Chip Node Architecture
    • Interconnect Architecture
    • Heterogeneous Multicore Chip Architecture (Not covered in this lecture)
What Are the Issues? Parallelism!

Map from the models/types of parallel computation of algorithms, programming languages and applications to the models and types of parallelism of hardware architectures.

- **Core** – SISD, SIMD
- **Node** – MIMD/shared
- **Interconnect** – MIMD/distributed
What are the Issues? Memory Hierarchy!

- Multiple levels of memory
  - Five levels of caching
- Complex Parallelism in Memory Access
  - Cache Line Fetches
- Interference among processors for access to memory which is shared among processors
- Effects of prefetching
What Knowledge is Needed to Map Applications to Architectures?

• Characteristics of Memory Hierarchy
  – Core
  – Chip
  – Node
  – Interconnect

• Parallelism:
  – Algorithms/Libraries
  – Resource Control – Multicore Programming
  – Programming Models/Languages
  – Hardware Architectures
Ranger Architecture - Single Processor Architecture

Processor Core for AMD Barcelona Chip
Ranger Architecture - Chip Architecture

Balanced, Highly Efficient Cache Structure

- **Dedicated L1**
  - Locality keeps most critical data in the L1 cache
  - Lowest latency
  - 2 loads per cycle

- **Dedicated L2**
  - Sized to accommodate the majority of working sets today
  - Dedicated to eliminate conflicts common in shared caches
  - Better for Virtualization

- **Shared L3 - NEW**
  - Victim-cache architecture maximizes efficiency of cache hierarchy
  - Fills from L3 leave likely shared lines in the L3
  - Sharing-aware replacement policy
  - Ready for expansion at the right time for customers

AMD Barcelona Chip
Ranger Architecture - Node Architecture
Ranger - Node Memory Architecture

- 32 GB per node
- Assume 32K pages => ~1,000,000 pages/node
- 48 pages mapped in TLB
- DRAM Bank – Parallel accessibility to each bank
- DRAM pages - A DRAM page ((not to be confused with OS pages.) ) represents a row of data that has been read from a DRAM bank and is cached within the DRAM for faster access. DRAM pages can be large, 32 KB in the case of Ranger, and there are typically two per DIMM. This means that a Ranger node shares 32 different 32 KB pages among 16 cores on four chips, yielding a megabyte of SRAM cache in the main memory.
Ranger Interconnect - InfiniBand

Properties: One Hop Connectivity, Packets and DMA between Nodes
Why Do Applications Care about Architecture?

• Return on Investment
  – Understand What You are Doing!
  – Performance
    • Choice of Algorithms
    • Choice of Libraries
    • Optimization of Application Specific Code
  – Productivity
    • Personal
    • Resource
What Do Application Developers/Users Need to Know about Architecture?

• Parallelism in Software
  – Algorithm
  – Resource Control
  – Language/Library/Programming Model
  – Hardware Architecture

• Parallelism in Cluster Computer Architecture
  – Processor/Core Architecture
  – Homogeneous Multicore Chips
  – Node Architecture
  – Interconnect Architecture
  – Heterogeneous Multicore Chips (Not covered in this lecture.)
Flynn’s Classic Parallel Hardware Taxonomy

Processor Organizations

- Single instruction, single data (SISD) stream
  - Uniprocessor
    - Vector processor
    - Array processor
- Single instruction, multiple data (SIMD) stream
- Multiple instruction, single data (MISD) stream
- Multiple instruction, multiple data (MIMD) stream
  - Shared memory
  - Distributed memory
    - Clusters
      - Symmetric multiprocessor (SMP)
      - Nonuniform memory access (NUMA)
Amdahl and Gustafson’s Laws

Amdahl’s Law – Speedup S for:
F = fraction parallelizable and
P = number of processors
assuming perfect parallelism and identical problem size.

\[ S = \frac{1}{(1-F) + \frac{F}{P}} \]

\[ S = \frac{1}{1-F} \]

Gustafson’s Law – Speedup S for
P(n) = fraction parallelizable as a function of problem size measured by n and P = number of processors assuming perfect parallelism but increasing problem size.

\[ S = (1-F(n)) + P \times F(n) \]

\[ S = P \times F(n) \]

In some problems, F(n) \( \Rightarrow 1 \) as n becomes large so that S \( \Rightarrow P \).
Types of Parallelism

• Streaming Parallelism
• Vector Parallelism
• Asynchronous SIMD => SPMD
• Synchronous SIMD
• Asynchronous MIMD
• Shared Name Space Parallel Execution
• Distributed Name Space Parallel Execution
Taxonomy of Parallelism

Attributes { 
• mechanism for control – synchronous (implicit by time), asynchronous (explicit) 
• resolution of control – static, dynamic 
• cardinality of control – single instruction stream, multiple instruction stream 
• locality of control – single, distributed 
• state data for control – central/shared, distributed 
• name space for execution data – shared, distributed 
• cardinality of input/output (data) stream – single, multiple 
• granularity of data access – discrete, chunk, stream 
• }

Models - {control: (mechanism, resolution, cardinality, locality, state data), name space, cardinality of input/output, granularity of data access}
SIMD \{(synchronous, static, multiple, single, central), shared, multiple, stream\}

Naturally suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
MISD - \{(synchronous, static, multiple, single, central), shared, single, stream\} – MISD

• Natural uses include:
  – multiple frequency filters operating on a single signal stream
  – multiple cryptography algorithms attempting to crack a single coded message.
MIMD - \{(asynchronous, static/dynamic, multiple, single, central/distributed), multiple, discrete\} – MIMD

• Most Clusters, Servers and Grids
Visible Parallelism

• Invisible Parallelism – Internal to core or instruction
  – Prefetching from Memory
  – Pipelining of instruction execution
  – SIMD instructions

• Mostly addressed by compilers and runtime systems but compilers often need help.

• Will be addressed in later lectures on performance optimization.
Visible Parallelism

• Four Levels of Visible Parallelism
  – Core
  – Chip
  – Node
  – System

• Generally asynchronous and user controllable.
Threads – Software and Hardware

• Software
  – Multiple operating system defined threads (pThreads) can be created on each core.
  – The thread scheduler can switch among them but only one can be active simultaneously.

• But some core/chip architecture have hardware implemented threads.
  – Simultaneous Multithreading
Multi-core: threads can run on separate cores
Multi-core: multiple threads can run on each core
A technique complementary to multi-core: Simultaneous multithreading on one core

- Problem addressed:
  The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

Other execution units wait unused

Source: Intel
Simultaneous multithreading (SMT)

• Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
• Weaving together multiple “threads” on the same core

• Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units
Without SMT, only a single thread can run at any given time
Without SMT, only a single thread can run at any given time

Thread 2: integer operation
SMT processor: both threads can run concurrently

- Thread 1: floating point
- Thread 2: integer operation

Flowchart showing the architecture of the SMT processor with

- L1 D-Cache D-TLB
- Integer
- Floating Point
- Schedulers
- Uop queues
- Rename/Alloc
- BTB
- Trace Cache
- Decoder
- BTB and I-TLB
- L2 Cache and Control
- Bus
- uCode ROM
But: Can’t simultaneously use the same functional unit

This scenario is impossible with SMT on a single core (assuming a single integer unit)
SMT not a “true” parallel processor

• Enables better threading (e.g. up to 30%)
• OS and applications perceive each simultaneous thread as a separate “virtual processor”
• The chip has only a single copy of each resource
• Compare to multi-core: each core has its own copy of resources
Combining Multi-core and SMT

• Cores can be SMT-enabled (or not)
• The different combinations:
  – Single-core, non-SMT: standard uniprocessor
  – Single-core, with SMT
  – Multi-core, non-SMT
  – Multi-core, with SMT: our fish machines
• The number of SMT threads:
  2, 4, or sometimes 8 simultaneous threads
• Intel calls them “hyper-threads”
SMT Dual-core: all four threads can run concurrently
Comparison: multi-core vs SMT

• Multi-core:
  – Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  – However, great with thread-level parallelism

• SMT
  – Can have one large and fast superscalar core
  – Great performance on a single thread
  – Mostly still only exploits instruction-level parallelism
Node Level Parallelism

• Memory sharing and interference issues become even more complex.
• Ranger nodes have asymmetries which lead to additional issues of load imbalance and cores getting out of “phase.”
• I/O is at node level – Each node can be writing to the file system simultaneously and asynchronously.
The memory hierarchy

• If simultaneous multithreading only:
  – all caches shared

• Multi-core chips:
  – L1 caches core private
  – L2 caches core private in some architectures and shared in others
  – L3 caches usually shared across cores
  – DRAM page cache usually shared

• Memory is always shared
Designs with private L2 caches

Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D

A design with L3 caches

Example: Intel Itanium 2
Private vs shared caches

• Advantages of private:
  – They are closer to core, so faster access
  – Reduces contention

• Advantages of shared:
  – Threads on different cores can share the same cache data
  – More cache space available if a single (or a few) high-performance thread runs on the system
The cache coherence problem

• Since we have private caches: How to keep the data consistent across caches?
• Each core should perceive the memory as a monolithic array, shared by all the cores
The cache coherence problem

Suppose variable x initially contains 15213

Core 1
One or more levels of cache

Core 2
One or more levels of cache

Core 3
One or more levels of cache

Core 4
One or more levels of cache

Main memory
x=15213

multi-core chip
The cache coherence problem

Core 1 reads x

- Core 1: One or more levels of cache, x=15213
- Core 2: One or more levels of cache
- Core 3: One or more levels of cache
- Core 4: One or more levels of cache

Main memory, x=15213

Multi-core chip
The cache coherence problem

Core 2 reads $x$

- Core 1: One or more levels of cache $x=15213$
- Core 2: One or more levels of cache $x=15213$
- Core 3: One or more levels of cache
- Core 4: One or more levels of cache

Main memory $x=15213$
The cache coherence problem

Core 1 writes to $x$, setting it to 21660

- Core 1: One or more levels of cache $x=21660$
- Core 2: One or more levels of cache $x=15213$
- Core 3: One or more levels of cache
- Core 4: One or more levels of cache

Main memory $x=21660$

multi-core chip

assuming write-through caches
The cache coherence problem

Core 2 attempts to read x... gets a stale copy

[Diagram showing a multi-core chip with interconnected cores and levels of cache]
Solutions for cache coherence

• This is a general problem with multiprocessors, not limited just to multi-core
• There exist many solution algorithms, coherence protocols, etc.
• We will return to consistency and coherence issues in a later lecture.

• A HARDWARE solution: invalidation-based protocol with snooping
Inter-core bus

![Diagram of inter-core bus with cores connected by inter-core bus](image-url)
Invalidation protocol with snooping

• Invalidation:
  If a core writes to a data item, all other copies of this data item in other caches are invalidated

• Snooping:
  All cores continuously “snoop” (monitor) the bus connecting the cores.
The cache coherence problem

Revisited: Cores 1 and 2 have both read x
The cache coherence problem

Core 1 writes to \( x \), setting it to 21660

- Core 1
  - One or more levels of cache
    - \( x = 21660 \)
- Core 2
  - One or more levels of cache
    - \( x = 15213 \)
- Core 3
  - One or more levels of cache
- Core 4
  - One or more levels of cache

 inval

Main memory
- \( x = 21660 \)

\{ assuming write-through caches \}

multi-core chip

inter-core bus

sends invalidation request

\textbf{INVALIDATED}
The cache coherence problem

After invalidation:

- Core 1: One or more levels of cache
  - x=21660
- Core 2: One or more levels of cache
- Core 3: One or more levels of cache
- Core 4: One or more levels of cache

Main memory:
  - x=21660

multi-core chip
The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.

Core 1
One or more levels of cache
x=21660

Core 2
One or more levels of cache
x=21660

Core 3
One or more levels of cache

Core 4
One or more levels of cache

Main memory
x=21660

multi-core chip
Invalidation protocols

• This was just the basic invalidation protocol
• More sophisticated protocols use extra cache state bits
• MSI, MESI - (Modified, Exclusive, Shared, Invalid) to which we will return later
**Another hardware solution: update protocol**

Core 1 writes $x=21660$:

- Core 1
  - One or more levels of cache
    - $x=21660$
- Core 2
  - One or more levels of cache
    - $x=21660$
- Core 3
  - One or more levels of cache
- Core 4
  - One or more levels of cache

Main memory
  - $x=21660$

UPDATED broadcasts updated value

assuming write-through caches

multi-core chip

inter-core bus
Invalidation vs update

• Multiple writes to the same location
  – invalidation: only the first time
  – update: must broadcast each write
    (which includes new variable value)

• Invalidation generally performs better:
  it generates less bus traffic
User Control of Multi-core Parallelism

• Programmers must use threads or processes

• Spread the workload across multiple cores

• Write parallel algorithms

• OS will map threads/processes to cores
Thread safety very important

• Pre-emptive context switching: context switch can happen AT ANY TIME

• True concurrency, not just uniprocessor time-slicing

• Concurrency bugs exposed much faster with multi-core
However: Need to use synchronization even if only time-slicing on a uniprocessor

```c
int counter=0;

void thread1() {
    int temp1=counter;
    counter = temp1 + 1;
}

void thread2() {
    int temp2=counter;
    counter = temp2 + 1;
}
```
Need to use synchronization even if only time-slicing on a uniprocessor

temp1 = counter;
counter = temp1 + 1;
temp2 = counter;
counter = temp2 + 1

gives counter = 2

temp1 = counter;
temp2 = counter;
counter = temp1 + 1;
counter = temp2 + 1

gives counter = 1
Assigning threads to the cores

• Each thread/process has an affinity mask

• Affinity mask specifies what cores the thread is allowed to run on

• Different threads can have different masks

• Affinities are inherited across fork()
Affinity masks are bit vectors

• Example: 4-way multi-core, without SMT

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>

• Process/thread is allowed to run on cores 0, 2, 3, but not on core 1
Affinity masks when multi-core and SMT combined

• Separate bits for each simultaneous thread
• Example: 4-way multi-core, 2 threads per core

Core 2 can’t run the process
Core 1 can only use one simultaneous thread
Default Affinities

- Default affinity mask is all 1s: all threads can run on all processors
- Then, the OS scheduler decides what threads run on what core
- OS scheduler detects skewed workloads, migrating threads to less busy processors
Process migration is costly

- Need to restart the execution pipeline
- Cached data is invalidated
- OS scheduler tries to avoid migration as much as possible:
  it tends to keeps a thread on the same core
- This is called *soft affinity*
User Set Affinities

• The programmer can prescribe her own affinities (hard affinities)
• Rule of thumb: use the default scheduler unless you have knowledge of program behavior
• There are also memory affinity tags which can be set.
When to set your own affinities

• Two (or more) threads share data-structures in memory
  – map to same core so that can share cache

• Real-time threads:
Example: a thread running a robot controller:
  - must not be context switched, or else robot can go unstable
  - dedicate an entire core just to this thread

Source: Sensable.com
Connections to Succeeding Lectures

• Parallel programming for multicore chips and multichip nodes.
• Concurrent/Parallel execution and consistency and coherence
• Models for thinking about and formulating parallel computations
• Performance optimization – Software engineering
Assignment

• To be submitted on Friday, 9/16 by 5PM
• Go to the User Guide for Ranger
  http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide
• Read the sections on affinity policy and NUMA Control
• Take the program which will be distributed on Wednesday, 9/14 and define the affinity policy and NUMA control policy you would use and justify your choices.