CS 207 D

Computer Architecture

Lecture 9: Multiprocessors
Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?
  - a. 10%
  - b. 5%
  - c. 1%
  - d. <1%
Amdahl’s Law Answers

- Speedup\_overall = \frac{1}{1 - \text{Fraction\_enhanced} + \text{Fraction\_enhanced}}

\[80 = \frac{1}{1 - \text{Fraction\_parallel} + \text{Fraction\_parallel}}\]

\[80 = \frac{1}{1 - \text{Fraction\_parallel} + \text{Fraction\_parallel}} = \frac{100}{100} \times \left\{ 1 - \text{Fraction\_parallel} + \text{Fraction\_parallel} \right\} = 1\]

\[79 = 80 \times \text{Fraction\_parallel} - 0.8 \times \text{Fraction\_parallel}\]

Fraction\_parallel = \frac{79}{79.2} = 99.75\%
Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local
- accesses hit memory hierarchy and base CPI is 0.5. (Remote access = $200/0.5 = 400$ clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?
  - a. 1.5X
  - b. 2.0X
  - c. 2.5X
CPI Equation

- CPI = Base CPI + Remote request rate x Remote request cost
  - CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
  - No communication (the MP with all local reference) is 1.3/0.5
  - or 2.6 faster than 0.2% instructions involve remote access
Challenges of Parallel Processing

1. Application parallelism primarily via new algorithms that have better parallel performance

2. Long remote latency impact both by architect and by the programmer
   - For example, reduce frequency of remote accesses either by
     - Caching shared data (HW)
     - Restructuring the data layout to make more accesses local (SW)
Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors
- inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data
  - reduces latency to shared data, memory bandwidth for shared data,
  - and interconnect bandwidth
  - cache coherence problem
Amdahl’s Law

- Architecture design is very bottleneck-driven
- make the common case fast
- do not waste resources on a component that has little impact on overall performance/power
- Amdahl’s Law:
  - performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Execution Time After Improvement = Execution Time Unaffected + Execution Time Affected
- Speedup = \( \frac{\text{Performance after improvement}}{\text{Performance before improvement}} \)
Amdahl's Law

- Example:
- Suppose a program runs in 100 seconds on a machine, with multiply responsible for 80 seconds of this time.
- How much do we have to improve the speed of the multiplication if we want the program to run 4 times faster?
- \[ 100 / 4 = 80/n + 20 \]
- \[ 5 = 80/n \]
- \[ n = 16 \] the multiply needs to be 16 times faster!
Exploiting Parallelism

- Most operations do not depend on each other
- Execute them in parallel
- At the circuit level, simultaneously access multiple ways
  of a set-associative cache
- At the organization level, execute multiple instructions at the same time
- At the system level, execute a different program while one is waiting on I/O!
Flynn Classification of parallel processing architectures

- Objective
- Be familiar with Flynn classification of parallel processing architectures
  - SISD, SIMD, MISD, MIMD
- Basic multiprocessor architectures
Flynn’s Taxonomy
• Flynn classified by data and control streams in 1966

<table>
<thead>
<tr>
<th>Single Instruction Single Data (SISD) (Uniprocessor)</th>
<th>Single Instruction Multiple Data SIMD (single PC: Vector, CM-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Instruction Single Data (MISD) (ASIP)</td>
<td>Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)</td>
</tr>
</tbody>
</table>

SIMD ⇐ Data Level Parallelism

• MIMD ⇐ Thread Level Parallelism

• MIMD popular because
  – Flexible: N pgms and 1 multithreaded pgm
  – Cost-effective: same MPU in desktop & MIMD
Flynn Classification

- SISD (single instruction and single data stream)
  - SIMD (single instruction and multiple data streams)
  - MISD (Multiple instructions and single data stream)
  - MIMD (Multiple instructions and multiple data streams)
SISD

- No instruction parallelism
- • No data parallelism
- • SISD processing architecture example—a personal computer processing instructions and data on single processor
**SIMD**

- Multiple data streams in parallel with a single instruction stream
  - • SIMD processing architecture example—a graphic processor processing instructions for translation or rotation or other operations are done on multiple data
  - • An array or matrix is also processed in SIMD

**MISD**

- Multiple instruction streams in parallel operating on single instruction stream
  - • Processing architecture example—processing for critical controls of missiles where single data stream processed on different processors to handle faults if any during processing
Multiple processing streams in parallel processing on parallel data streams

- MIMD processing architecture example is super computer or distributed computing systems with distributed or single shared memory
What is Parallel Architecture

- A parallel computer is a collection of processing elements that cooperate to solve large problems
- Most important new element: it is all about communication!!
- What does the programmer (or OS or compiler writer) think about?
- Models of computation
- Sequential consistency?
- Resource allocation
- What mechanisms must be in hardware
- A high performance processor (SIMD, or Vector Processor)
- Data access, Communication, and Synchronization
“A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”

- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:
  - 1. Centralized Memory Multiprocessor
    - <few dozen cores
    - Small enough to share single, centralized memory with uniform memory access latency (UMA)
  - 2. Physically Distributed-Memory Multiprocessor
    - Larger number chips and cores than 1.
    - BW demands Memory distributed among processors with non-uniform memory access/latency (NUMA)
Multicore: Mainstream Multiprocessors

Multicore chips
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- **AMD Quad Phenom**
  - Four 2+ GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- **Intel Core 2 Quad**
  - Four cores, shared 4 MB L2
  - Two 4MB L2 caches
- **Sun Niagara**
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
- For servers, not desktop

All have same shared memory programming model
But First, Uniprocessor Concurrency

• Software “thread”
• Independent flow of execution
• Context state: PC, registers
• Threads generally share the same memory space
• “Process” like a thread, but different memory space
• Java has thread support built in, C/C++ supports P-threads library
• Generally, system software (the O.S.) manages threads
• “Thread scheduling”, “context switching”
• All threads share the one processor
• Hardware timer interrupt occasionally triggers O.S.
• Quickly swapping threads gives illusion of concurrent execution
• Much more in CIS380
Aside: Hardware Multithreading

Hardware Multithreading (MT)
- Multiple threads dynamically share a single pipeline (caches)
- Replicate thread contexts: PC and register file
- **Coarse-grain MT**: switch on L2 misses Why?
- **Simultaneous MT**: no explicit switching, fine-grain interleaving
- Pentium4 is 2-way hyper-threaded, leverages out-of-order core

+ MT Improves utilization and throughput
- Single programs utilize <50% of pipeline (branch, cache miss)
- MT does not improve single-thread performance
- Individual threads run as fast or even slower
Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism

- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model

- **Multiprocessing**
  - Multiply execution resources, higher peak performance
  - Some interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores
Example: Parallelizing Matrix Multiply

for (I = 0; I < 100; I++)
for (J = 0; J < 100; J++)
for (K = 0; K < 100; K++)

• How to parallelize matrix multiply over 100 processors?
• One possibility: give each processor 100 iterations of I
  for (J = 0; J < 100; J++)
  for (K = 0; K < 100; K++)
• Each processor runs copy of loop above
• my_id() function gives each processor ID from 0 to N
• Parallel processing library provides this function

CIS 501 (Martin/Roth): Multicore
GPU Performance

Graphics Processing Units (GPUs) have been evolving at a rapid rate in recent years.
CPU Performance

- CPUs have also been increasing functional unit counts
- But with a lot more complexity
  - Reorder buffers/reservations stations
  - Complex branch prediction
- This means that CPUs add raw compute power at a much slower rate
GPU vs. CPU

- Disparity is largely due to the specific nature of problems historically solved by the GPU
  - Same operations on many primitives (SIMD)
  - Focus on throughput over latency
  - Lots of special purpose hardware
- CPUs
  - Focus on reducing Latency
  - Designed to handle a wider range of problems
GPU Limitations

- Relatively small amount of memory, < 4GB in current GPUs
- I/O directly to GPU memory has complications
  - Must transfer to host memory, and then back
  - If 10% of instructions are LD/ST and other instructions are...
    » 10 times faster \( \frac{1}{0.1 + 0.9/10} \) speedup of 5
    » 100 times faster \( \frac{1}{0.1 + 0.9/100} \) speedup of 9
Programming GPUs

Advantages
– Supercomputer-like FP performance on commodity processors

Disadvantages
– Performance tuning difficult
– Large speed gap between compiler-generated and hand-tuned code

CMSC
Matrix Multiplication Example

**Original Fortran**

```fortran
do i = 1,n
  do j = 1,m
    do k = 1,p
      a(i,j) = a(i,j) + b(i,k)*c(k,j)
    enddo
  enddo
enddo
```
Introduction to reconfigurable logic and special-purpose processors

- The basic principle of a system making use of reconfigurable logic
Systems provided of reconfigurable logic are often called Reconfigurable Instruction Set Processors (RISP). The reconfigurable logic includes a set of programmable processing units, which can be reconfigured in the field to implement logic operations or functions, and programmable interconnections between them.
To execute a program taking advantage of the reconfigurable logic, usually the following steps are necessary (illustrated in Fig. 2.3):

1. Code Analysis: the first thing to do is to identify parts of the code that can be transformed for execution on the reconfigurable logic. The goal of this step is to find the best tradeoff considering performance and available resources regarding
Basic steps in a reconfigurable system

**STEP 1**: Code is analyzed. Hot spots are found.

**STEP 2**: Hot spots are transformed to reconfigurable instructions.

**STEP 3**: A reconfigurable instruction is found. Reconfiguration begins and the logic is set.

**STEP 4**: Input operands are fetched.

**STEP 5**: The configuration is executed.

**STEP 6**: Results are written back.
Reconfiguration Steps

2- Code transformation: Once the best candidate parts of code to be accelerated are found.

3- Reconfiguration After code transformation, it is time to send to the reconfigurable system.

4. Input Context Loading: To perform a given reconfigurable operation, a set of

5. Execution: After the reconfigurable unit is set and the proper input operands

6. Write back: The results of the reconfigurable operation are saved back to the register file,
Underlying Execution Mechanism

- Different ways of performing the same computation
• Programming multiprocessor systems

What is concurrency?

● What is a sequential program?
A single thread of control that executes one instruction and when it is finished execute the next logical instruction

● What is a concurrent program?
A collection of autonomous sequential threads, executing (logically) in parallel

● The implementation (i.e. execution) of a collection of threads can be: Multiprogramming – Threads multiplex their executions on a single processor. Multiprocessing – Threads multiplex their executions on a multiprocessor or a multicore system Distributed Processing – Processes multiplex their executions on several different machines
**Why use Concurrent Programming?**

- Natural Application Structure
  - The world is not sequential! Easier to program multiple independent and concurrent activities.
- Increased application throughput and responsiveness
- Not blocking the entire application due to blocking IO
- Performance from multiprocessor/multicore hardware
- Parallel execution
- Distributed systems
- Single application on multiple machines
- Client/server type or peer-to-peer systems
Comparing Uniprocessing to Multiprocessing

- A Uniprocessor (UP) system has one processor. A driver in a UP system executes one thread at any given time. Either a kernel thread (processed by the upper half of a driver), or on the Interrupt Control Stack (ICS) in a processor interrupt context (processed by the lower half of a driver). The UP driver synchronization model coordinates execution between the driver’s upper and lower halves.
Comparing Uniprocessing to Multiprocessing

- A Multiprocessor (MP) system has two or more processors. A driver in this system can execute multiple threads concurrently. For each processor, the driver may be executing in a kernel thread or in the interrupt context of that processor. The MP driver synchronization model coordinates execution among multiple kernel threads as well as between the driver’s upper and lower halves.
- HP-UX is a multiprocessing operating system, its drivers must utilize synchronization mechanisms designed for an MP system. Drivers that use these mechanisms will work correctly on both MP and UP systems.