Readings: Multiprocessing

- **Required**

- **Recommended**
Memory Consistency

- Required
Readings: Cache Coherence

**Required**

**Recommended:**
- Culler and Singh, *Parallel Computer Architecture*
  - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
- P&H, *Computer Organization and Design*
  - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
Multiprocessors and Issues in Multiprocessing
Remember: Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

- Main (or Original) Goal
  - Improve performance (Execution time or task throughput)
    - Execution time of a program governed by Amdahl’s Law

- Other Goals
  - Reduce power consumption
    - (4N units at freq F/4) consume less power than (N units at freq F)
    - Why?
  - Improve cost efficiency and scalability, reduce complexity
    - Harder to design a single unit that performs as well as N simpler units
  - Improve dependability: Redundant execution in space
Types of Parallelism and How to Exploit Them

- **Instruction Level Parallelism**
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- **Data Parallelism**
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- **Task Level Parallelism**
  - Different “tasks/threads” can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)
Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively

- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task
Multiprocessing Fundamentals
Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization
Main Design Issues in Tightly-Coupled MP

- Shared memory synchronization
  - How to handle synchronization: locks, atomic operations, barriers

- Cache coherence
  - How to ensure correct operation in the presence of private caches keeping the same memory address cached

- Memory consistency: Ordering of all memory operations
  - What should the programmer expect the hardware to provide?

- Shared resource management

- Communication: Interconnects
Main Programming Issues in Tightly-Coupled MP

- **Load imbalance**
  - How to partition a single task into multiple tasks

- **Synchronization**
  - How to synchronize (efficiently) between tasks
  - How to communicate between tasks
  - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, ...

- **Contention (avoidance & management)**
- **Maximizing parallelism**
- **Ensuring correct operation while optimizing for performance**
Aside: Hardware-based Multithreading

- **Coarse grained**
  - Quantum based
  - Event based (switch-on-event multithreading), e.g., switch on L3 miss

- **Fine grained**
  - Cycle by cycle

- **Simultaneous**
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization
Fine-Grained Multithreading

- **Idea:** Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
  - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread.
  - Branch/instruction resolution latency overlapped with execution of other threads’ instructions.

+ No logic needed for handling control and data dependences within a thread.
  - Single thread performance suffers.
  - Extra logic for keeping thread contexts.
  - Does not overlap latency if not enough threads to cover the whole pipeline.

[Video Link](https://www.youtube.com/watch?v=6e5KZcCGBYw&list=PL5Q2soXY2Zj_uej3aY39YB5pfW4SJ7LIN&index=16)
More on Multithreading (I)

Coarse-grained Multithreading

- Idea: When a thread is stalled due to some event, switch to a different hardware context
- Switch-on-event multithreading


1,252 views • Nov 19, 2013

Carnegie Mellon Computer Architecture
23K subscribers

Lecture 9: Multithreading
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu)
Date: September 25, 2013.

https://www.youtube.com/onurmutlulectures
More on Multithreading (II)

Carnegie Mellon - Parallel Computer Architecture 2012 - Onur Mutlu - Lecture 10 - Multithreading II

1,594 views · Sep 21, 2013

Carnegie Mellon Computer Architecture

Lecture 10: Multithreading II
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 28, 2012.

https://www.youtube.com/onurmutlulectures
More on Multithreading (III)
More on Multithreading (IV)

Now that We Have MT Hardware …
- what else can we use it for?
- Redundant execution to tolerate soft (and hard?) errors
- Implicit parallelization: thread level speculation
  - Slippstream processors
  - Leader-follower architectures
- Helper threading
  - Prefetching
  - Branch prediction
- Exception handling

Panorama.png

https://www.youtube.com/onurmutlulectures
Lectures on Multithreading

- Parallel Computer Architecture, Fall 2012, Lecture 9
  - Multithreading I (CMU, Fall 2012)
  - https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=51

- Parallel Computer Architecture, Fall 2012, Lecture 10
  - Multithreading II (CMU, Fall 2012)
  - https://www.youtube.com/watch?v=e8lfl6MbILg&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=52

- Parallel Computer Architecture, Fall 2012, Lecture 13
  - Multithreading III (CMU, Fall 2012)
  - https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=53

- Parallel Computer Architecture, Fall 2012, Lecture 15
  - Speculation I (CMU, Fall 2012)
  - https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=54

https://www.youtube.com/onurmutlulectures
Limits of Parallel Speedup
Parallel Speedup Example

- \(a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0\)

- Assume given inputs: \(x\) and each \(a_i\)

- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?
Single processor: 11 operations (data flow graph)
\[ R = a_4 x^4 + a_3 x^2 + a_2 x^2 + a_1 x + a_0 \]

Three processors: \( T_3 \) (execute with 3 proc.)

\[ T_3 = 5 \text{ cycles} \]
Speedup with 3 Processors

\[ \tau_3 = \frac{5 \text{ cycles}}{} \]

Speedup with 3 processors = \( \frac{11}{5} = 2.2 \)

\[ \left( \frac{\tau_1}{\tau_3} \right) \]

Is this a fair comparison?
Revisiting the Single-Processor Algorithm

Revisit T₁

Better single-processor algorithm:

\[ R = a₄x^4 + a₃x^3 + a₂x^2 + a₁x + a₀ \]

\[ R = (((a₄x + a₃)x + a₂)x + a₁)x + a₀ \]

(Horner's method)

\[ I_1 = 8 \text{ cycles} \]

Speedup with 3 procs.

\[
\text{Speedup} = \frac{t_1^{\text{best}}}{t_3^{\text{best}}} = \frac{8}{5} = 1.6
\]

\( \text{Note:} \) 2.2
Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- **Unfair comparisons**
  Compare best parallel algorithm to wimpy serial algorithm → unfair

- **Cache/memory effects**
  More processors → more cache or memory → fewer misses in cache/mem
Utilization, Redundancy, Efficiency

- **Traditional metrics**
  - Assume all P processors are tied up for parallel computation

- **Utilization**: How much processing capability is used
  - \( U = \frac{\text{# Operations in parallel version}}{\text{processors} \times \text{Time}} \)

- **Redundancy**: how much extra work is done with parallel processing
  - \( R = \frac{\text{# of operations in parallel version}}{\text{# operations in best single processor algorithm version}} \)

- **Efficiency**
  - \( E = \frac{\text{Time with 1 processor}}{\text{processors} \times \text{Time with P processors}} \)
  - \( E = \frac{U}{R} \)
Utilization of a Multiprocessor

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Utilization**: How much processing capability we use.

\[
U = \frac{\text{Ops with p proc.}}{p \times T_p}
\]

Example:

\[
U = \frac{10 \text{ operations (in parallel version)}}{3 \text{ processors} \times 5 \text{ time units}} = \frac{10}{15}
\]
Redundancy: How much extra work due to multiprocessing

\[ R = \frac{\text{Ops with } p \text{ proc.}^{\text{best}}}{\text{Ops with } 1 \text{ proc.}^{\text{best}}} = \frac{10}{8} \]

\[ R \text{ is always } \geq 1 \]

Efficiency: How much resource we use compared to how much resource we can get away with

\[ E = \frac{1 \cdot T_t^{\text{best}}}{p \cdot T_p} \quad (\text{tying up } 1 \text{ proc for } T_t \text{ time units}) \]

\[ = \frac{8}{15} \quad (E = \frac{U}{R}) \]
Amdahl’s Law and Caveats of Parallelism
Caveats of Parallelism (I)

Why the reality? (diminishing returns)

\[ T_p = \alpha \cdot \frac{T_1}{P} + (1 - \alpha) \cdot T_1 \]

- Parallelizable part/fixtures of the single-processor program
- Non-parallelizable part
Amdahl’s Law

\[
\text{Speedup with } p \text{ proc.} = \frac{T_1}{T_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}
\]

As \( p \to \infty \),

\[
\text{Speedup} = \frac{1}{1-\alpha} \quad \text{but the slowest for parallel Speedup}
\]

Amdahl’s Law Implication 1

Amdahl’s Law illustrated

Adding more and more processors gives less and less benefit; if $\alpha < 1$
The benefit (speedup) is small until $\alpha \approx 1$. 

Amdahl’s Law Implication 2
Caveats of Parallelism (II)

- Amdahl’s Law
  - \( f \): Parallelizable fraction of a program
  - \( N \): Number of processors

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}
\]


- Maximum speedup limited by serial portion: Serial bottleneck

- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among \( N \) processors)
Sequential Bottleneck

![Graph showing speedup vs parallel fraction for different N values: N=10, N=100, and N=1000. The graph demonstrates the impact of parallel fraction on speedup.]
Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: **Non-parallelizable operations on data** (e.g. non-parallelizable loops)
  
  ```
  for (i = 0; i < N; i++)
  ```

- There are other causes as well:
  - Single thread prepares data and spawns parallel tasks (usually sequential)
Another Example of Sequential Bottleneck (I)

InitPriorityQueue(PQ);
SpawnThreads();
ForEachThread:

while (problem not solved)
    Lock (X)
    SubProblem = PQ.remove();
    Unlock(X);
    Solve(SubProblem);
    If(problem solved) break;
    NewSubProblems = Partition(SubProblem);
    Lock(X)
    PQ.insert(NewSubProblems);
    Unlock(X)

PrintSolution();

Another Example of Sequential Bottleneck (II)

Bottlenecks in Parallel Portion

- **Synchronization**: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - **Communication**: Tasks may need values from each other
    - Causes thread serialization when shared data is contended

- **Load Imbalance**: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
    - Reduces speedup in parallel portion

- **Resource Contention**: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
    - Additional latency not present when each task runs alone
Bottlenecks in Parallel Portion: Another View

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications

- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, ...

- Some threads can be on the critical path of execution due to synchronization; some threads are not

- Even within a thread, some “code segments” may be on the critical path of execution; some are not
Remember: Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:

```plaintext
loop {
    Compute
    lock(A)
    Update shared data
    unlock(A)
}
```
Remember: Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving to the barrier is on the critical path

Each thread:

```cpp
loop1 {
    Compute
}
barrier
loop2 {
    Compute
}
```

![Diagram of barrier synchronization](image)
Remember: Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```plaintext
loop {
  Compute1
  Compute2
  Compute3
}
```
Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - “Embarrassingly parallel” applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of parallel computer architecture is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer’s job easier in writing correct and high-performance parallel programs
We Have Already Seen Examples
In Previous Two Lectures

- Lecture 17b: Parallelism and Heterogeneity
  - https://www.youtube.com/watch?v=GLzG_rEDn9A&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=18

- Lecture 18a: Bottleneck Acceleration
  - https://www.youtube.com/watch?v=P8l3SMAbyYw&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=19
More on Accelerated Critical Sections

- M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures" 
  One of the 13 computer architecture papers of 2009 selected as Top Picks by IEEE Micro.

Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman  
University of Texas at Austin  
suleman@hps.utexas.edu

Onur Mutlu  
Carnegie Mellon University  
onur@cmu.edu

Moinuddin K. Qureshi  
IBM Research  
mkquresh@us.ibm.com

Yale N. Patt  
University of Texas at Austin  
patt@ece.utexas.edu
More on Bottleneck Identification & Scheduling

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications"

Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao  
ECE Department  
The University of Texas at Austin  
joao@ece.utexas.edu

M. Aater Suleman  
Calxeda Inc.  
aater.suleman@calxeda.com

Onur Mutlu  
Computer Architecture Lab.  
Carnegie Mellon University  
onur@cmu.edu

Yale N. Patt  
ECE Department  
The University of Texas at Austin  
patt@ece.utexas.edu
More on Utility-Based Acceleration

- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"

Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (ppt) Slides (pdf)

Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao † M. Aater Suleman ‡‡ Onur Mutlu § Yale N. Patt †

† ECE Department
The University of Texas at Austin
Austin, TX, USA
{joao, patt}@ece.utexas.edu

‡ Flux7 Consulting
Austin, TX, USA
suleman@hps.utexas.edu

§ Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu
More on Data Marshaling


One of the 11 computer architecture papers of 2010 selected as Top Picks by IEEE Micro.

Data Marshaling for Multi-core Architectures

M. Aater Suleman† Onur Mutlu§ José A. Joao† Khubaib† Yale N. Patt†

†The University of Texas at Austin
{su, joa, kh, patt}@hps.utexas.edu

§Carnegie Mellon University
onur@cmu.edu
An Example Parallel Problem: Task Assignment to Processors
Static versus Dynamic Scheduling

- **Static**: Done at compile time or parallel task creation time
  - Schedule does not change based on runtime information

- **Dynamic**: Done at run time (e.g., after tasks are created)
  - Schedule changes based on runtime information

- Example: Instruction scheduling
  - Why would you like to do dynamic scheduling?
  - What pieces of information are not available to the static scheduler?
Parallel Task Assignment: Tradeoffs

Problem: $N$ tasks, $P$ processors, $N > P$. Do we assign tasks to processors statically (fixed) or dynamically (adaptive)?

**Static assignment**
- Simpler: No movement of tasks.
- Inefficient: Underutilizes resources when load is not balanced

**Dynamic assignment**
- Efficient: Better utilizes processors when load is not balanced
- More complex: Need to move tasks to balance processor load
- Higher overhead: Task movement takes time, can disrupt locality
Parallel Task Assignment: Example

- Compute histogram of a large set of values
- Parallelization:
  - Divide the values across $T$ tasks
  - Each task computes a local histogram for its value set
  - Local histograms merged with global histograms in the end

```c
GetPageHistogram(Page *P)
{
    For each thread: {
        /* Parallel part of the function */
        UpdateLocalHistogram(Fraction of Page)

        /* Serial part of the function */
        Critical Section:
            Add local histogram to global histogram
        Barrier
    }

    Return global histogram
}
```
Parallel Task Assignment: Example (II)

- How to schedule tasks updating local histograms?
  - Static: Assign equal number of tasks to each processor
  - Dynamic: Assign tasks to a processor that is available
  - When does static work as well as dynamic?

- Implementation of Dynamic Assignment with Task Queues

(a) Distributed Task Stealing
(b) Hierarchical Task Queuing
Software Task Queues

- What are the advantages and disadvantages of each?
  - Centralized
  - Distributed
  - Hierarchical

(a) Distributed Task Stealing
(b) Hierarchical Task Queuing
Task Stealing

- **Idea:** When a processor’s task queue is empty it steals a task from another processor’s task queue
  - Whom to steal from? (Randomized stealing works well)
  - How many tasks to steal?

+ Dynamic balancing of computation load

- Additional communication/synchronization overhead between processors
- Need to stop stealing if no tasks to steal
Parallel Task Assignment: Tradeoffs

- Who does the assignment? Hardware versus software?

Software
  + Better scope
  - More time overhead
  - Slow to adapt to dynamic events (e.g., a processor becoming idle)

Hardware
  + Low time overhead
  + Can adjust to dynamic events faster
  - Requires hardware changes (area and possibly energy overhead)
How Can the Hardware Help?

- Managing task queues in software has overhead
  - Especially high when task sizes are small

- An idea: Hardware Task Queues
  - Each processor has a dedicated task queue
  - Software fills the task queues (on demand)
  - Hardware manages movement of tasks from queue to queue
  - There can be a global task queue as well → hierarchical tasking in hardware

  - Optional reading
Dynamic Task Generation

- Does static task assignment work in this case?

- Problem: Searching the exit of a maze

```plaintext
while (problem not solved)
    SubProblem = PriorityQ.remove()
    Solve(SubProblem)
    if (solved)
        break
    NewSubProblems = Partition(SubProblem)
    PriorityQ.insert(NewSubProblems)
```
Programming Model vs. Hardware Execution Model
Programming Models vs. Architectures

- Five major models
  - (Sequential)
  - Shared memory
  - Message passing
  - Data parallel (SIMD)
  - Dataflow
  - Systolic

- Hybrid models?
Shared Memory vs. Message Passing

- Are these programming models or execution models supported by the hardware architecture?

- Does a multiprocessor that is programmed by “shared memory programming model” have to support a shared address space processors?

- Does a multiprocessor that is programmed by “message passing programming model” have to have no shared address space between processors?
Programming Models: Message Passing vs. Shared Memory

- **Difference:** how communication is achieved between tasks

  - **Message passing programming model**
    - Explicit communication via messages
    - Loose coupling of program components
    - Analogy: telephone call or letter, no shared location accessible to all

  - **Shared memory programming model**
    - Implicit communication via memory operations (load/store)
    - Tight coupling of program components
    - Analogy: bulletin board, post information at a shared space

- **Suitability of the programming model depends on the problem to be solved. Issues affected by the model include:**
  - Overhead, scalability, ease of programming, bugs, match to underlying hardware, ...
Message Passing vs. Shared Memory Hardware

- **Difference:** how task communication is supported in hardware

  - **Shared memory hardware (or machine model)**
    - All processors see a global shared address space
      - Ability to access all memory from each processor
    - A write to a location is visible to the reads of other processors

  - **Message passing hardware (machine model)**
    - No global shared address space
    - Send and receive variants are the only method of communication between processors (much like networks of workstations today, i.e. clusters)

- Suitability of the hardware depends on the problem to be solved as well as the programming model.
Programming Model vs. Hardware

- Most of parallel computing history, there was no separation between programming model and hardware
  - Message passing: Caltech Cosmic Cube, Intel Hypercube, Intel Paragon
  - Shared memory: CMU C.mmp, Sequent Balance, SGI Origin.
  - SIMD: ILLIAC IV, CM-1

- However, any hardware can really support any programming model

- Why?
  - Application → compiler/library → OS services → hardware