Today and Next Week

- Multiprocessors
- Memory Consistency
- Cache Coherence
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 locks, atomic operations

- **Cache coherence**
  - How to ensure correct operation in the presence of private caches

- **Memory consistency: Ordering of 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, ...

- **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
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?
\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

Single processor: 11 operations (data flow graph)

\[ \tau_1 = 11 \text{ cycles} \]
\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

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

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

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

\[ \text{Speedup with 3 processors} = \frac{11}{5} = 2.2 \]

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

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

Better single-processor algorithm:

\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

\[ R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0 \]

(Horner's method)

$T_1 = 8$ cycles

\[
\text{Speedup with 3 procs.} = \frac{T_1}{T_{\text{best}}} = \frac{8}{5} = 1.6
\]
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

Multiprocessor metrics.

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

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

\[ U = \frac{10 \text{ operations (in parallel version)}}{3 \text{ processors} \times 5 \text{ time units}} = \frac{10}{15} \]
冗余：因多处理而增加的工作量

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

\( R \) 总是 \( \geq 1 \)

效率：我们使用了多少资源与我们可以避开的资源的比较

\[ E = \frac{1. T_1^{\text{best}}}{p \cdot T_p^{\text{best}}} \quad (\text{tieg up 1 proc for } T_p \text{ time units}) \]

\[ = \frac{8}{15} \quad \left( E = \frac{U}{R} \right) \]
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/fraction of the single-processor program

non-parallelizable part
Amdahl’s Law

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

\[
\text{Speedup as } p \to \infty = \frac{1}{1-\alpha} \quad \text{bottleneck for parallel Speedup}
\]

Amdahl’s Law Implication 1

Adding more and more processors gives less and less benefit if $\alpha < 1$.

$\alpha = 0.98$
$\alpha = 0.95$
$\alpha = 0.9$
Amdahl’s Law Implication 2

The benefit (speedup) is small until $\alpha \approx 1$. 

$\text{Speedup}$

$\alpha$

$P_1$

$P_2$

$P_3$
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, N=1000).]
Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)
  
  ```c
  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)

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
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:
```
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:

```java
loop1 {
    Compute
}
barrier
loop2 {
    Compute
}
```
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

```
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

- Heterogeneous Multi-Core Systems
  - [https://www.youtube.com/watch?v=UC_ROevjIuM](https://www.youtube.com/watch?v=UC_ROevjIuM)

- Bottleneck Acceleration
  - [https://www.youtube.com/watch?v=-4eNBfz1Eqk](https://www.youtube.com/watch?v=-4eNBfz1Eqk)
More on Accelerated Critical Sections

More on Bottleneck Identification & Scheduling


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)
More on Bottleneck Identification & Scheduling


Data Marshaling for Multi-core Architectures

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

†The University of Texas at Austin
{suleman, joao, khubaib, patt}@hps.utexas.edu

§Carnegie Mellon University
onur@cmu.edu