Compsci 590.3:
Introduction to Parallel Computing

Alvin R. Lebeck

Slides based on those from the University of Oregon
Admin

Logistics
• No Office hours today
• Talk by D.E. Shaw
• Homework 2
• Homework 3 Wednesday (maybe)
• Projects
  – Groups may be different
  – Lightning proposal presentation

Outline
• Performance scalability
• Analytical performance measures
• Amdahl’s law and Gustafson-Barsis’ law
• Tuning
• Caches (if time, whiteboard)

Guest Lectures
• Mon Sept 21: Nikos Pitsianis
  – Biotonic Sort
• Mon Nov 2: Lars Nyland (NVIDIA)
Performance and Scalability

- **Evaluation**
  - Sequential runtime ($T_{seq}$) is a function of
    - problem size and architecture
  - Parallel runtime ($T_{par}$) is a function of
    - problem size and parallel architecture
    - # processors used in the execution
  - Parallel performance affected by
    - algorithm + architecture

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
  - Actually many ways to define scalability, so beware
    - Constant problem
    - Constant time
    - Constant error
Performance Metrics and Formulas

- $T_1$ is the execution time on a single processor
- $T_p$ is the execution time on a $p$ processor system

- $S(p)$ ($S_p$) is the speedup
  \[ S(p) = \frac{T_1}{T_p} \]

- Efficiency
  \[ \text{Efficiency} = \frac{S_p}{p} \]

- $Cost(p)$ ($C_p$) is the cost
  \[ Cost = p \times T_p \]

- Parallel algorithm is cost-optimal
  - Parallel time = sequential time ($C_p = T_1$, $E_p = 100\%$)
Amdahl’s Law (Fixed Size Speedup)

- Let $f$ be the fraction of a program that is sequential
  - $1-f$ is the fraction that can be parallelized
- Let $T_1$ be the execution time on 1 processor
- Let $T_p$ be the execution time on $p$ processors
- $S_p$ is the speedup
  \[
  S_p = \frac{T_1}{T_p} = \frac{T_1}{(fT_1 + (1-f)T_1/p)} = \frac{1}{(f + (1-f)/p)}
  \]
- As $p \to \infty$, $S_p = 1/f$
Amdahl’s Law and Scalability

- **Scalability**
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- **When does Amdahl’s Law apply?**
  - When the problem size is fixed
  - *Strong scaling* ($p \to \infty$, $S_p = S_\infty \to 1 / f$)
  - Speedup bound is determined by the degree of sequential execution time in the computation, not # processors!!!
  - Uhh, this is not good … Why?
  - Perfect efficiency is hard to achieve
    - Also means we still need very good sequential performance!
Gustafson-Barsis’ Law and Scalability

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

- When does Gustafson’s Law apply?
  - When the problem size can increase as the number of processors increases
  - *Weak scaling* \((S_p = 1 + (p-1)f_{par})\)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales
Amdahl versus Gustafson-Baris

Amdahl

serial work

parallelizable work

Time

P=1

P=2

P=4

P=8
Amdahl versus Gustafson-Baris

Gustafson-Baris

parallelizable work

serial work

Time

P=1 P=2 P=4 P=8
Reasoning about Performance

- How do we reason about performance for a parallel program on a given set of processors?
Recall: DAG Model of Computation

- Think of a program as a directed acyclic graph (DAG) of tasks
  - A task can not execute until all the inputs to the tasks are available
  - These come from outputs of earlier executing tasks
  - DAG shows explicitly the task dependencies

- Think of the hardware as consisting of workers (processors)

- Consider a greedy scheduler of the DAG tasks to workers
  - No worker is idle while there are tasks still to execute
Different: DAG Model of Computation

- What is fastest each program can execute on 1, inf, 2 processors?
  - Number is time for each task.
Work-Span Model

- \( TP = \) time to run with \( P \) workers
- \( T_1 = \) work
  - Time for serial execution
    - execution of all tasks by 1 worker
  - Sum of all work
- \( T_\infty = \) span
  - Time along the critical path
- Critical path: Sequence of task execution (path) through DAG that takes the longest time to execute
  - Assumes an infinite # workers available
Work-Span Example

- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can the execute in any order?
- $T_\infty = 5$
  - Time along the *critical path*
  - In this case, it is the longest pathlength of any task order that maintains necessary dependencies
Lower/Upper Bound on Greedy Scheduling

- Suppose we only have \( P \) workers
- We can write a work-span formula to derive a lower bound on \( T_P \)
  - \( \text{Max}(T_1 / P, T_{\infty}) \leq T_P \)
- \( T_{\infty} \) is the best possible execution time

- Brent’s Lemma derives an upper bound
  - Capture the additional cost executing the other tasks not on the critical path
  - \( T_P \leq \text{time for non-critical tasks} + \text{time for critical path} \)
  - Assume can do so without overhead
  - \( T_P \leq (T_1 - T_{\infty}) / P + T_{\infty} \)
Consider Brent’s Lemma for 2 Processors

- $T_1 = 7$
- $T_\infty = 5$
- $T_2 \leq (T_1 - T_\infty) / P + T_\infty$
  
  \[ \leq (7 - 5) / 2 + 5 \]
  
  \[ \leq 6 \]
Amdahl was an optimist!

![Graph showing speedup with different bounds: Amdahl's Law, Work-Span Bound, Brent's Lemma. The graph demonstrates the speedup as a function of the number of processors (P).]
Estimating Running Time

- Scalability requires that $T_\infty$ be dominated by $T_1$

\[ T_P \approx \frac{T_1}{P} + T_\infty \text{ if } T_\infty << T_1 \]

- Increasing work hurts parallel execution proportionately
- The span (critical path) impacts scalability, even for finite $P$
Parallel Slack

- Sufficient parallelism implies linear speedup

\[ T_p \approx \frac{T_1}{P} \text{ if } \frac{T_1}{T_\infty} \gg P \]

- Linear speedup
- Parallel slack
Asymptotic Complexity

- Time complexity of an algorithm summarizes how the execution time grows with input size.
- Space complexity summarizes how memory requirements grow with input size.
- Standard work-span model considers only computation, not communication or memory.
- Asymptotic complexity is a strong indicator of performance on large-enough problem sizes and reveals an algorithm’s fundamental limits.
Scalable Parallel Computing

- Scalability in parallel architecture
  - Processor numbers
  - Memory architecture
  - Interconnection network
  - Avoid critical architecture bottlenecks
- Scalability in computational problem
  - Problem size
  - Computational algorithms
    - Computation to memory access ratio
    - Computation to communication ratio
- Parallel programming models and tools
- Performance scalability
Why Aren’t Parallel Applications Scalable?

- Sequential performance
- Critical Paths
  - Dependencies between computations spread across processors
- Bottlenecks
  - One processor holds things up
- Algorithmic overhead
  - Some things just take more effort to do in parallel
- Communication overhead
  - Spending increasing proportion of time on communication
- Load Imbalance
  - Makes all processor wait for the “slowest” one
  - Dynamic behavior
- Speculative loss
  - Do A and B in parallel, but B is ultimately not needed
Critical Paths

- Long chain of dependence
  - Main limitation on performance
  - Resistance to performance improvement

- Diagnostic
  - Performance stagnates to a (relatively) fixed value
  - Critical path analysis

- Solution
  - Eliminate long chains if possible
  - Shorten chains by removing work from critical path
Bottlenecks

- How to detect?
  - One processor A is busy while others wait
  - Data dependency on the result produced by A

- Typical situations:
  - N-to-1 reduction / computation / 1-to-N broadcast
  - One processor assigning job in response to requests

- Solution techniques:
  - More efficient communication
  - Hierarchical schemes for master slave

- Program may not show ill effects except for large # of processors
Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce additional operations (Why?)
  - *Parallel overhead*
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it doesn’t parallelize well (e.g., not scalable)
- What to do?
  - Choose algorithmic variants that minimize overhead
  - Use two level algorithms
- Performance is the rub
  - Are you achieving better parallel performance?
  - Must compare with the best sequential algorithm
Factors which determine a program's performance are complex, interrelated, and sometimes hidden.

Application related factors
  - Algorithms dataset sizes, task granularity, memory usage patterns, load balancing, I/O communication patterns.

Hardware related factors
  - Processor architecture, memory hierarchy, I/O network.

Software related factors
  - Operating system, compiler/preprocessor, communication protocols, libraries.
Utilization of Computational Resources

- Resources can be under-utilized or used inefficiently
  - Identifying these circumstances can give clues to where performance problems exist
- Resources may be “virtual”
  - Not actually a physical resource (e.g., thread, process)
- Performance analysis tools are essential to optimizing an application's performance
  - Can assist you in understanding what your program is "really doing"
  - May provide suggestions how program performance should be improved
Performance Analysis and Tuning: The Basics

- Most important goal of performance tuning is to reduce a program's wall clock execution time
  - Iterative process to optimize efficiency
  - Efficiency is a relationship of execution time

- So, where does the time go?

- Find your program's hot spots and eliminate the bottlenecks in them
  - **Hot spot**: an area of code within the program that uses a disproportionately high amount of processor time
  - **Bottleneck**: an area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays

- Understand *what, where, and how* time is being spent
Parallel Performance Engineering Process

Implementation

Preparation

Performance Analysis

Program Tuning

Production

Measurement

Analysis

Ranking

Refinement
Sequential Performance

- Sequential performance is all about:
  - How time is distributed
  - What resources are used where and when

- “Sequential” factors
  - Computation
    - choosing the right algorithm is important
    - compilers can help
  - Memory systems and cache and memory
    - more difficult to assess and determine effects
    - modeling can help
  - Input / output
Parallel Performance

- Parallel performance is about sequential performance AND parallel interactions
  - Sequential performance is the performance within each thread of execution
  - “Parallel” factors lead to overheads
    - concurrency (threading, processes)
    - communication (message passing or caching effects)
    - synchronization (both explicit and implicit)
  - Parallel interactions also lead to parallelism inefficiency
    - load imbalances
Sequential Performance Tuning

- Sequential performance tuning is a *time-driven* process
- Find the thing that takes the most time and make it take less time (i.e., make it more efficient)
- May lead to program restructuring
  - Changes in data storage and structure
  - Rearrangement of tasks and operations
- May look for opportunities for better resource utilization
  - Cache management is a big one
  - Locality, locality, locality!
  - Virtual memory management may also pay off
- May look for opportunities for better processor usage
Parallel Performance Tuning

- In contrast to sequential performance tuning, parallel performance tuning might be described as *conflict-driven* or *interaction-driven*.
- Find the points of parallel interactions and determine the overheads associated with them.
- Overheads can be the cost of performing the interactions:
  - Transfer of data
  - Extra operations to implement coordination
- Overheads also include time spent waiting:
  - Lack of work
  - Waiting for dependency to be satisfied
Interesting Performance Phenomena

- Superlinear speedup
  - Speedup in parallel execution is greater than linear
  - $S_p > p$
  - How can this happen?

- Need to keep in mind the relationship of performance and resource usage

- Computation time (i.e., real work) is not simply a linear distribution to parallel threads of execution

- Resource utilization thresholds can lead to performance inflections