CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Performance Analysis

Instructor: Haidar M. Harmanani
Spring 2018

Outline

- Performance scalability
- Analytical performance measures
- Amdahl’s law and Gustafson-Barsis’ law
Performance

- In computing, performance is defined by 2 factors
  - Computational requirements (what needs to be done)
  - Computing resources (what it costs to do it)
- Computational problems translate to requirements
- Computing resources interplay and tradeoff

\[
\text{Performance} \sim \frac{1}{\text{Resources for solution}}
\]

- Hardware
- Time
- Energy
- ... and ultimately
- Money

Measuring Performance

- Performance itself is a measure of how well the computational requirements can be satisfied
- We evaluate performance to understand the relationships between requirements and resources
  – Decide how to change “solutions” to target objectives
- Performance measures reflect decisions about how and how well “solutions” are able to satisfy the computational requirements
- When measuring performance, it is important to understand exactly what you are measuring and how you are measuring it
Scalability

- A program can scale up to use many processors
  - What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation
  - If double the number of processors, what to expect?
  - Is scalability linear?
- Use parallel efficiency measure
  - Is efficiency retained as problem size increases?
- Apply performance metrics

Performance and Scalability

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

- Scalability
  - Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem
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
  \[
  S(p) = \frac{T_1}{T_p}
  \]
- $E(p)$ ($E_p$) is the efficiency
  \[
  Efficiency = \frac{S_p}{p}
  \]
- $Cost(p)$ ($C_p$) is the cost
  \[
  Cost = p \times T_p
  \]

- Parallel algorithm is cost-optimal
  - Parallel time = sequential time ($C_p = T_1$, $E_p = 100\%$)

---

Speed-Up

- Provides a measure of application performance with respect to a given program platform
- Speedup can also be cast in terms of computational steps
  - Can extend time complexity to parallel computations
- Use the fastest known sequential algorithm for running on a single processor
What is a “good” speedup?

- Hopefully, $S(n) > 1$
- Linear speedup:
  - $S(n) = n$
  - Parallel program considered perfectly scalable
- *Superlinear* speedup:
  - $S(n) > n$
  - Can this happen?

Defining Speed-Up

- We need more information to evaluate speedup:
  - What problem size? Worst case time? Average case time?
  - What do we count as work?
    - Parallel computation, communication, overhead?
  - What serial algorithm and what machine should we use for the numerator?
    - Can the algorithms used for the numerator and the denominator be different?
Common Definitions of Speed-Up

- Common definitions of Speedup:
  - Serial machine is one processor of parallel machine and serial algorithm is interleaved version of parallel algorithm
    \[ S(n) = \frac{T(1)}{T(n)} \]
  - Serial algorithm is fastest known serial algorithm for running on a serial processor
    \[ S(n) = \frac{T_s}{T(n)} \]
  - Serial algorithm is fastest known serial algorithm running on a one processor of the parallel machine
    \[ S(n) = \frac{T'(1)}{T(n)} \]

Can speedup be superlinear?

- Speedup CANNOT be superlinear:
  - Let M be a parallel machine with n processors
  - Let T(X) be the time it takes to solve a problem on M with X processors
  - Speedup definition:
    \[ S(n) = \frac{T(1)}{T(n)} \leq \frac{nt}{t} = n \]
  - Suppose a parallel algorithm A solves an instance I of a problem in t time units
    - Then A can solve the same problem in \( n \times t \) units of time on M through time slicing
    - The best serial time for I will be no bigger than \( n \times t \)
    - Hence speedup cannot be greater than n.
Can speedup be superlinear?

- **Speedup CAN be superlinear:**
  - Let $M$ be a parallel machine with $n$ processors
  - Let $T(X)$ be the time it takes to solve a problem on $M$ with $X$ processors
  - Speedup definition: $S(n) = \frac{T_s}{T(n)}$
  - Serial version of the algorithm may involve more overhead than the parallel version of the algorithm
    - E.g. $A=B+C$ on a SIMD machine with $A,B,C$ matrices vs. loop overhead on a serial machine
  - Hardware characteristics may favor parallel algorithm
    - E.g. if all data can be decomposed in main memories of parallel processors vs. needing secondary storage on serial processor to retain all data
  - “work” may be counted differently in serial and parallel algorithms

### Speedup Factor

- Maximum speedup is usually $n$ with $n$ processors (linear speedup).
- Possible to get *superlinear* speedup (greater than $n$) but usually a specific reason such as:
  - Extra memory in multiprocessor system
  - Nondeterministic algorithm
Maximum Speedup: Amdahl’s law

- \( f \) = fraction of program (algorithm) that is serial and cannot be parallelized
- Data setup
- Reading/writing to a single disk file
- Speedup factor is given by:

\[
T_s = fT_s + (1 - f)T_p \\
T_p = fT_s + \left(1 - \frac{1}{n}\right)T_s \\
S(n) = \frac{T_s}{fT_s + \left(1 - \frac{1}{n}\right)T_s} = \frac{n}{1 + (n-1)f} \\
\lim_{n \to \infty} = \frac{1}{f}
\]

The above equation is known as Amdahl’s Law.
Note that as \( n \to \infty \), the maximum speedup is limited to \( 1/f \).

Bounds on Speedup

(a) One processor

(b) Multiple processors

\( t_p = (1 - f)t_s/p \)
Speedup Against Number of Processors

- Even with infinite number of processors, maximum speedup limited to $1/f$.
- Example: With only 5% of computation being serial, maximum speedup is 20, irrespective of number of processors.

Example of Amdahl’s Law (1)

- Suppose that a calculation has a 4% serial portion, what is the limit of speedup on 16 processors?
  - $16/(1 + (16 - 1) \times 0.04) = 10$
  - What is the maximum speedup?
    - $1/0.04 = 25$
Example of Amdahl’s Law (2)

- 95% of a program’s execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

\[
\psi \leq \frac{1}{0.05 + (1 - 0.05)/8} \approx 5.9
\]

Example of Amdahl’s Law (3)

- 20% of a program’s execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

\[
\lim_{{p \to \infty}} \frac{1}{0.2 + (1 - 0.2)/p} = \frac{1}{0.2} = 5
\]
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!!!
  - Perfect efficiency is hard to achieve

- See original paper by Amdahl on course webpage
Variants of Speedup: Efficiency

- Efficiency: \( E(n) = \frac{S(n)}{n} \times 100\% \)
- Efficiency measures the fraction of time that processors are being used on the computation.
  - A program with linear speedup is 100% efficient.
- Using efficiency:
  - A program attains 89% efficiency with a serial fraction of 2%. Approximately how many processors are being used according to Amdahl’s law?

Efficiency

\[
\text{Efficiency} = \frac{\text{Sequential execution time}}{\text{Processors used} \times \text{Parallel execution time}}
\]

Efficiency = \frac{\text{Speedup}}{\text{Processors used}}
Limitations of Speedup

- Conventional notions of speedup don’t always provide a reasonable measure of performance.
- Questionable assumptions:
  - "work" in conventional definitions of speedup is defined by operation count:
    - communication more expensive than computation on current high-performance computers
  - best serial algorithm defines the least work necessary
    - for some languages on some machines, serial algorithm may do more work -- (loop operations vs. data parallel for example)
  - good performance for many users involves fast time on a sufficiently large problem; faster time on a smaller problem (better speedup) is less interesting
  - traditional speedup measures assume a “flat memory approximation”, i.e. all memory accesses take the same amount of time

“Flat Memory Approximation”

- “Flat memory Approximation” – all accesses to memory take the same amount of time
  - in practice, accesses to information in cache, main memory and peripheral memory take very different amounts of time.
Another Perspective

- We often use faster computers to solve larger problem instances
- Let’s treat time as a constant and allow problem size to increase with number of processors

Limitations of Speedup

- Gustafson challenged Amdahl’s assumption that the proportion of a program given to serial computations ($f$) and the proportion of a program given to parallel computations remains the same over all problem sizes.
  - For example, if the serial part is a loop initialization and it can be executed in parallel over the size of the input list, then the serial initialization becomes a smaller proportion of the overall calculation as the problem size grows larger.
- Gustafson defined two “more relevant” notions of speedup
  - Scaled speedup
  - Fixed-time speedup
    - (usual version he called fixed-size speedup)
Gustafson-Barsis’s Law

- Begin with parallel execution time
- Estimate sequential execution time to solve same problem
- Problem size is an increasing function of p
- Predicts scaled speedup

Gustafson’s Law

Fix execution time on a single processor
- $s + p = \text{serial part} + \text{parallelizable part} = 1$ (normalized serial time)
- $(s = \text{same as f previously})$
- Assume problem fits in memory of serial computer
- Fixed-size speedup

\[
S_{\text{fixed size}} = \frac{s + p}{s + \frac{p}{n}} = \frac{1}{s + \frac{1 - s}{n}}
\]

Amdahl’s law

Fix execution time on a parallel computer (multiple processors)
- $s + p = \text{serial part} + \text{parallelizable part} = 1$ (normalized parallel time)
- $(s + np = \text{serial time on a single processor})$
- Assume problem fits in memory of parallel computer
- Scaled Speedup

\[
S_{\text{scaled}} = \frac{s + np}{s + p} = n + (1 - n)s
\]
Scaled Speedup

- Scaling implies that problem size can increase with number of processors
  - Gustafson's law gives measure of how much
- Scaled Speedup derived by fixing the parallel execution time
  - Amdahl fixed the problem size → fixes serial execution time
  - Amdahl's law may be too conservative for high-performance computing.
- Interesting consequence of scaled speedup: no bound to speedup as $n \to \infty$, speedup can easily become superlinear!
- In practice, unbounded scalability is unrealistic as quality of answer will reach a point where no further increase in problem size may be justified

Meaning of Scalability Function

- To maintain efficiency when increasing $p$, we must increase $n$
- Maximum problem size limited by available memory, which is linear in $p$
- Scalability function shows how memory usage per processor must grow to maintain efficiency
- Scalability function a constant means parallel system is perfectly scalable
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

- See original paper by Gustafson on course webpage
Amdahl

serial work
parallelizable work

Time

P=1  P=2  P=4  P=8

Gustafson-Baris

serial work
parallelizable work

Time

P=1  P=2  P=4  P=8
Using Gustafson’s Law

- Given a scaled speedup of 20 on 32 processors, what is the serial fraction from Amdahl’s law? What is the serial fraction from Gustafson’s Law?

\[ S_{scaled} = \frac{s + np}{s + p} = n + (1 - n)s \]

Example 1

- An application running on 10 processors spends 3% of its time in serial code. What is the scaled speedup of the application?

\[ \psi = 10 + (1 - 10)(0.03) = 10 - 0.27 = 9.73 \]

...except 9 do not have to execute serial code

Execution on 1 CPU takes 10 times as long...
Example 2

- What is the maximum fraction of a program’s parallel execution time that can be spent in serial code if it is to achieve a scaled speedup of 7 on 8 processors?

\[ 7 = 8 + (1 - 8)s \Rightarrow s \approx 0.14 \]

Why Are not Parallel Applications Scalable?

<table>
<thead>
<tr>
<th>Critical Paths</th>
<th>Communication overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dependencies between computations spread across processors</td>
<td>Spending increasing proportion of time on communication</td>
</tr>
<tr>
<td>Bottlenecks</td>
<td>Load Imbalance</td>
</tr>
<tr>
<td>One processor holds things up</td>
<td>Makes all processor wait for the “slowest” one</td>
</tr>
<tr>
<td>Algorithmic overhead</td>
<td>Dynamic behavior</td>
</tr>
<tr>
<td>Some things just take more effort to do in parallel</td>
<td>Speculative loss</td>
</tr>
<tr>
<td></td>
<td>Do A and B in parallel, but B is ultimately not needed</td>
</tr>
</tbody>
</table>
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 for a long time
- Shows up when scaling
Algorithmic Overhead

- Different sequential algorithms to solve the same problem
- All parallel algorithms are sequential when run on 1 processor
- All parallel algorithms introduce addition operations
  - Parallel overhead
- Where should be the starting point for a parallel algorithm?
  - Best sequential algorithm might not parallelize at all
  - Or, it does not 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

What is the maximum parallelism possible?

- Depends on application, algorithm, program
  - Data dependencies in execution
  - Parallelism varies!
Embarrassingly Parallel Computations

- An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously
  - In a truly embarrassingly parallel computation there is no interaction between separate processes
  - In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way
- Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms
  - If it takes $T$ time sequentially, there is the potential to achieve $T/P$ time running in parallel with $P$ processors
  - What would cause this not to be the case always?

No or very little communication between processes
Each process can do its tasks without any interaction with other processes

Examples
- Numerical integration
- Mandelbrot set
- Monte Carlo methods
Calculating $\pi$ with Monte Carlo

Consider a circle of unit radius

Place circle inside a square box with side of 2 in

The ratio of the circle area to the square area is:

$$\frac{\pi \times 1 \times 1}{2 \times 2} = \frac{\pi}{4}$$

Monte Carlo Calculation of $\pi$

- Randomly choose a number of points in the square
- For each point $p$, determine if $p$ is inside the circle
- The ratio of points in the circle to points in the square will give an approximation of $\pi/4$
Using Programs to Measure Machine Performance

- Speedup measures performance of an individual program on a particular machine
  - Speedup cannot be used to
    - Compare different algorithms on the same computer
    - Compare the same algorithm on different computers

- Benchmarks are representative programs which can be used to compare performance of machines

Benchmarks used for Parallel Machines

- The Perfect Club
- The Livermore Loops
- The NAS Parallel Benchmarks
- The SPEC Benchmarks
- The “PACKS” (Linpack, LAPACK, ScaLAPACK, etc.)
- ParkBENCH
- SLALOM, HINT
Limitations and Pitfalls of Benchmarks

- Benchmarks cannot address questions you did not ask
- Specific application benchmarks will not tell you about the performance of other applications without proper analysis
- General benchmarks will not tell you all the details about the performance of your specific application
- One should understand the benchmark itself to understand what it tells us

Benefits of Benchmarks

- Popular benchmarks keep vendors attuned to applications
- Benchmarks can give useful information about the performance of systems on particular kinds of programs
- Benchmarks help in exposing performance bottlenecks of systems at the technical and applications level