LAU
الجامعة اللبنانية الأميركية
Lebanese American University

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

Performance Analysis

Instructor: Haidar M. Harmanani
Spring 2021

Today's Agenda

- Performance
- Scalability
- Benchmarking

Serial Performance

- Serial performance bottlenecks usually come from the sub optimal or chaotic utilization of hardware resources :
  - Memory random accesses
  - Branch coding
  - Sub optimal compilation

Parallel Performance

- Parallel performance bottlenecks usually come from the parallel design of your algorithm or parallel coding :
  - Excessive synchronization
  - Load imbalance

Serial vs Parallel

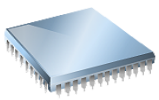
- Need to solve the serial problem before the parallel problem
  - Serial performance problems usually get a lot worse when run in parallel
  - The design of your parallel software depends on performance data collected in serial
  - Solving serial and parallel problems at the same time is too complex

Conclusion : solve your serial performance problems before you start parallelizing

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

$$Performance \sim \frac{1}{Resources\ for\ solution}$$


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 **problem size** and **architecture**
  - Parallel runtime (T_{par}) is a function of **problem size** and **parallel architecture** *and* the **number of 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

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 ($C_p = T_1$, Ep = 100%)

Speed-Up

- Provides a measure of application performance with respect to a given program platform
- 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

$$S(n) = \frac{T'(1)}{T(n)}$$

Parallel without performance is not enough

- Adding some parallelism to your software is often not enough to take advantage of many-core processors with efficiency and flexibility.
- Typical parallel performance issues :
  - Parallel overhead
  - Synchronization
  - Load imbalance
  - Granularity

Parallel Overhead

- All forms of parallelism bring a small overhead : loading a library, launching threads, scheduling …
- Solutions :
  - Monitor software and OS resources (memory usage, context switches, number of threads ...)
  - Remember that some parallel framework are light, designed for single computers and small task while others are very heavy, designed for large clusters.

Synchronization

- Some algorithms (like the blur filter) require communications, synchronizations between parallel executions, often blocking execution.
- Solutions :
  - Is another algorithm possible ?
  - Do you accept a slightly different result ?
  - Adapt your code to work with local variables ?
  - Optimize synchronization (→OpenMP course)

Load Imbalance

- Uneven distribution of chunks of data over the worker threads is a typical performance problem.
- Solutions :
  - Insert parallelism deeper in the call stack (pixels instead of files in our example)
  - Propose a new usage model for your software, easier to parallelize (in our example, process files in batch more easily)
  - Adapt the settings of your parallel framework (→ OpenMP algorithms and chunk size)

Granularity

- You have granularity problems if the chunks of data distributed to your threads are too big or too small. Too big they may cause a load imbalance. Too small a parallel overhead.
- Solutions :
  - Partition your data with flexibility. Hardware, data and usage models change rapidly.
  - Adapt the distribution algorithm and chunk size in your parallel framework.

Can speedup be superlinear?

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 n processors $S(n) = \frac{T(1)}{T(n)}$
  - 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 x t* units of time on M through time slicing
    - The best serial time for I will be no bigger than *n x 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

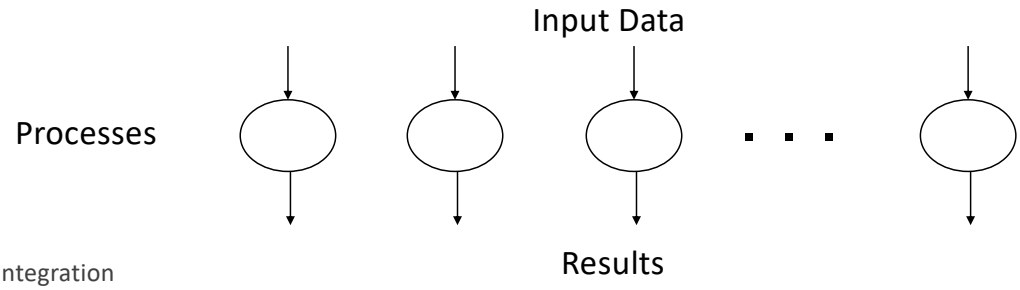
Embarrassingly Parallel Computations

- 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?

Embarrassingly Parallel Computations

No or very little communication between processes

Each process can do its tasks without any interaction with other processes



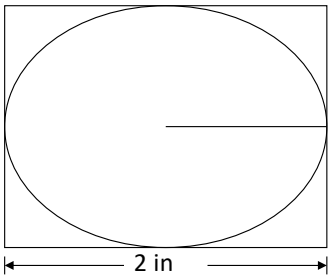
Examples

- Numerical integration
- Mandelbrot set
- Monte Carlo methods

Calculating π with Monte Carlo

Consider a circle of unit radius

Place circle inside a square box with side of 2 in

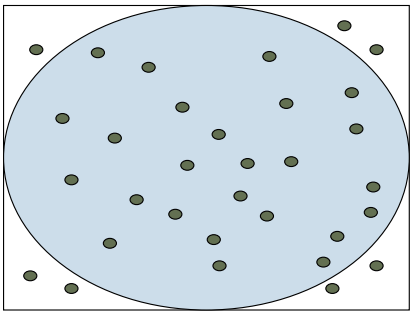


The ratio of the circle area to the square area is:

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

Monte Carlo Calculation of π

- 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 π/4



Empirical Performance Computation

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

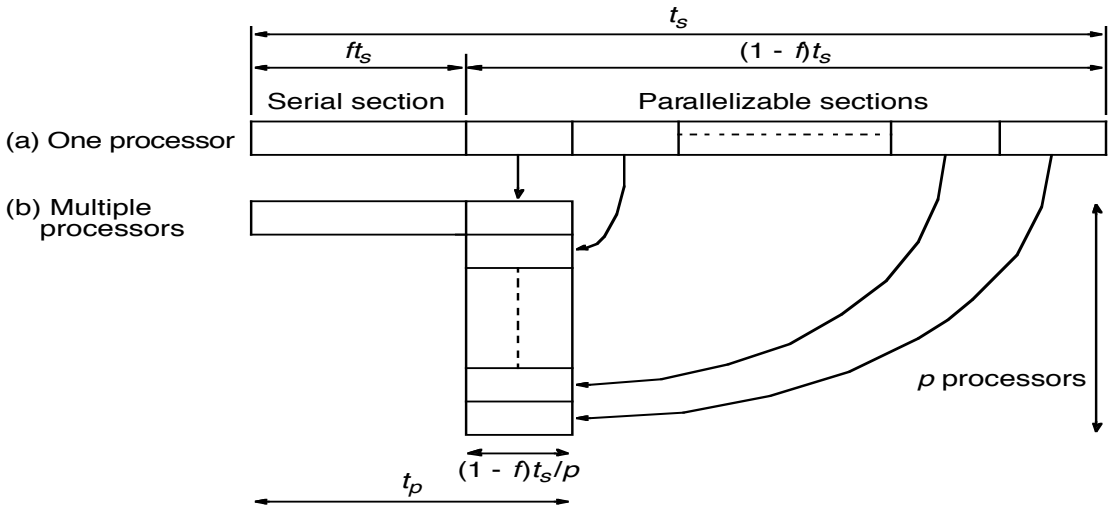
- Popular benchmarks keep vendors attuned to applications
- Benchmarks can give useful information about the performance of systems on particular kinds of programs
- Benchmarks help in exposing performance bottlenecks of systems at the technical and applications level

Theoretical Performance Computation

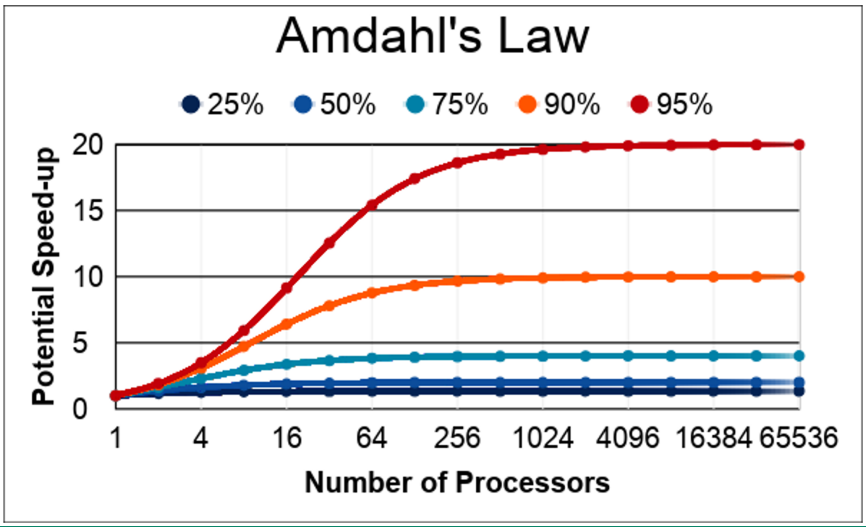
Amdahl's Law

- Serialization limits Performance
- Amdahl's law is an observation that the speed-up one gets from parallelizing the code is limited by the remaining serial part.
- Any remaining serial code will reduce the possible speed-up
- This is why it's important to focus on parallelizing the most time consuming parts, not just the easiest.

Amdahl's Law



Amdahl's Law



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:

$$T_s = fT_s + (1-f)T_s$$

$$T_p = fT_s + \frac{(1-f)T_s}{n}$$

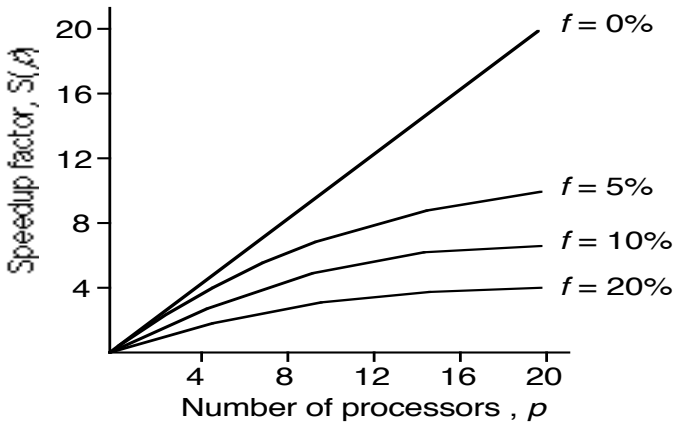
$$S(n) = \frac{T_s}{fT_s + \frac{(1-f)T_s}{n}} = \frac{n}{1+(n-1)f}$$

$$\lim_{n\to\infty} = \frac{1}{f}$$

Note that as n $\rightarrow \infty$, the maximum speedup is limited to 1/f.

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)*.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?

$$\psi \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

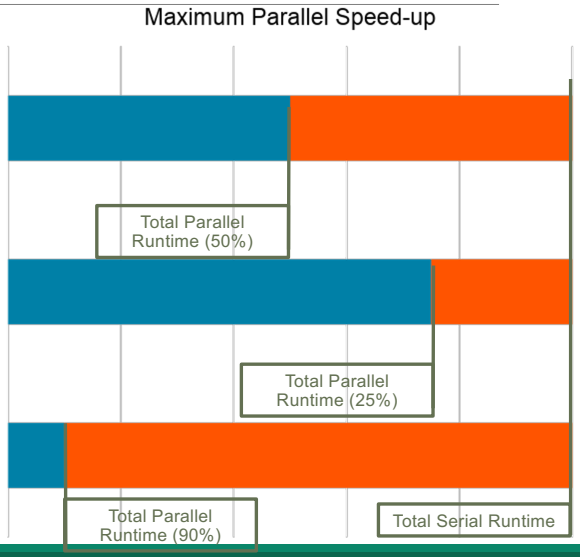
Example of Amdahl's Law (3)

- 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?

$$\lim_{p \to \infty} \frac{1}{0.2 + (1 - 0.2) / p} = \frac{1}{0.2} = 5$$

Example of Amdahl's Law (4)

- What's the maximum speed-up that can be obtained by parallelizing 50% of the code?
- (1 / 100% - 50%) = (1 / 1.0 - 0.50) = 2.0X
- What's the maximum speed-up that can be obtained by parallelizing 25% of the code?
- (1 / 100% - 25%) = (1 / 1.0 - 0.25) = 1.3X
- What's the maximum speed-up that can be obtained by parallelizing 90% of the code?
- (1 / 100% - 90%) = (1 / 1.0 - 0.90) = 10.0X



Variants of Speedup: Efficiency

- Efficiency: E(n) = S(n)/n * 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

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

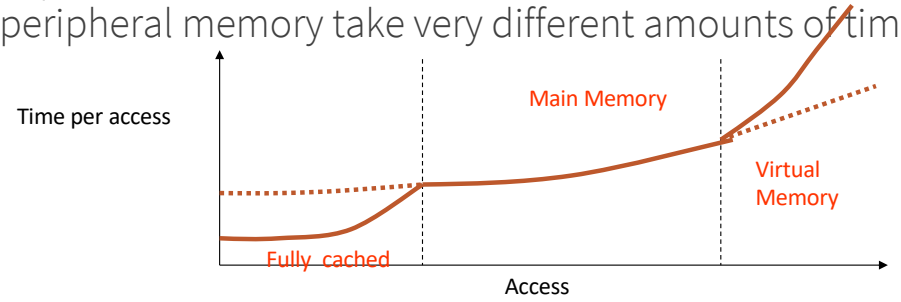
$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors used}}$$

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.



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

Another Perspective

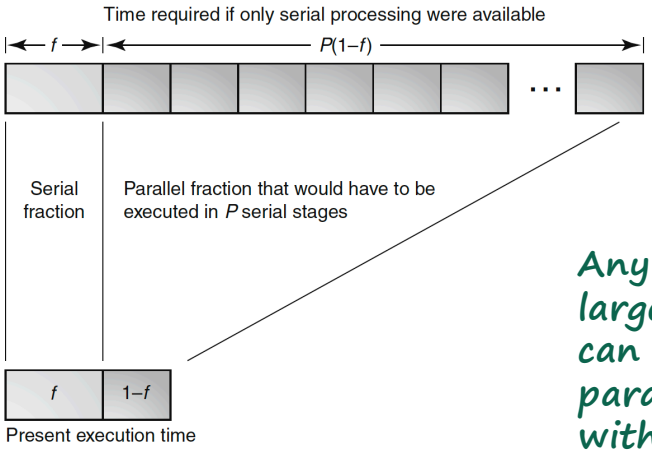
- We often use faster computers to solve larger problem instances
- Let's treat time as a constant and allow problem size to increase with number of processors

Limitations of Speedup

- Gustafson challenged Amdahl's assumption that the proportion of a program given to serial computations and the proportion of a program given to parallel computations remains the same over all problem sizes

[...] speedup should be measured by scaling the problem to the number of processors, not fixing problem size – John Gustafson

Gustafson-Barsis's Law



Gustafson's Law. Fig. 1 Graphical derivation of Gustafson's Law

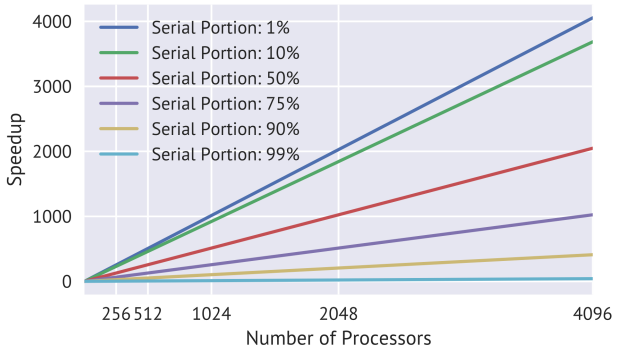
Any sufficiently large problem can be efficiently parallelized with a speedup

Limitations of Speedup

- Thus, 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)

Amdahl’s Law versus Gustafson’s Law

- Amdahl’s Law **fixes the problem size** and answers the question of *how parallel processing can reduce the execution time*
- Gustafson’s Law **fixes the run time** and answers the question of *how much longer time the present workload would take in the absence of parallelism*
  - $S = p - \alpha (p - 1)$
    - *P number of processors*
    - α *is the serial portion of the program*



Amdahl's Law versus Gustafson's Law

Fix execution time on a **single processor**

- $s + p$ = serial part + parallelizable part = **1** (normalized serial time)
- (s = same as *f* previously)
- Assume problem fits in memory of serial computer
- **Fixed-size speedup**

$$S_{fixed\_size} = \frac{s+p}{s+\frac{p}{n}} = \frac{1}{s+\frac{1-s}{n}}$$

Amdahl's law

Fix execution time on a **parallel computer (multiple processors)**

- $s + p$ = serial part + parallelizable part = **1** (normalized parallel time)
- $s + np$ = serial time on a single processor
- Assume problem fits in memory of parallel computer
- **Scaled Speedup**

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

Gustafson's Law

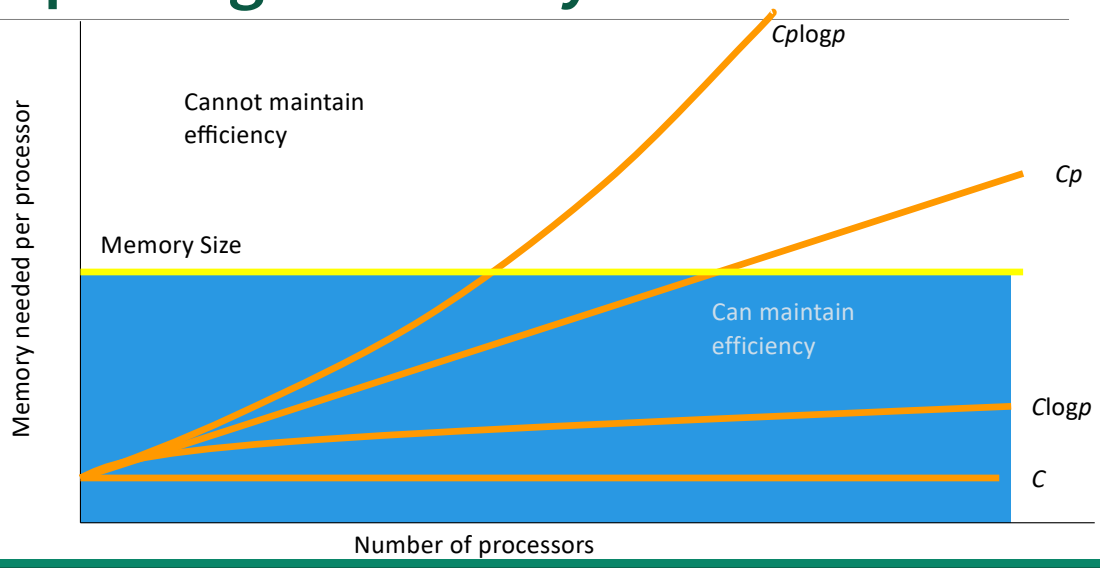
Scaled Speedup

- 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→ infinity, 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

Scalability

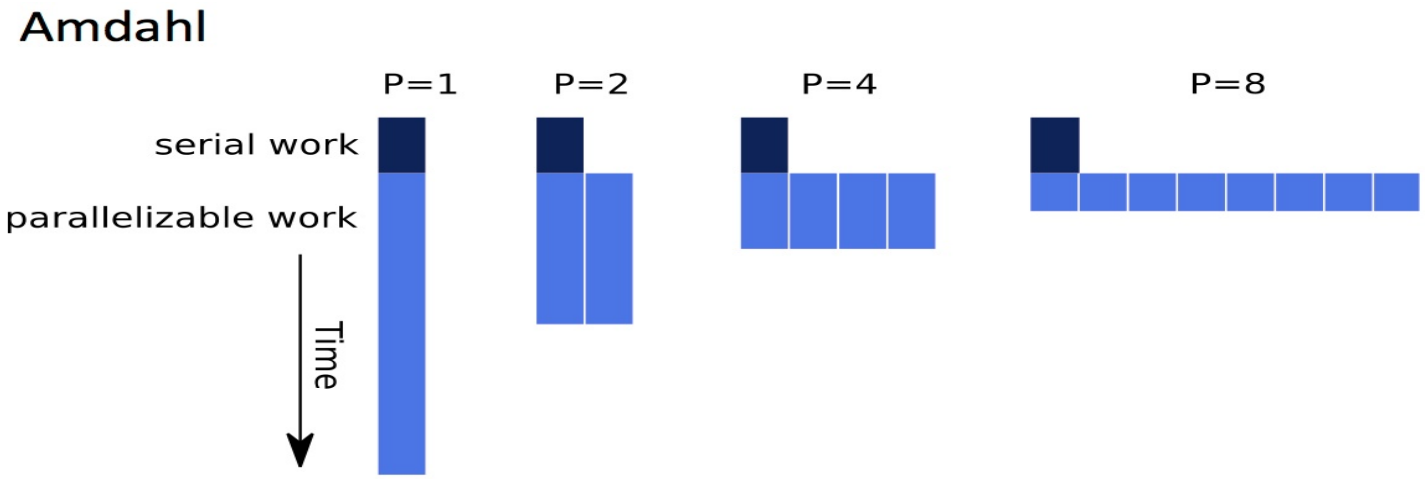
- Increase number of processors ➡️ decrease efficiency
- Increase problem size ➡️ increase efficiency
- Can a parallel system keep efficiency by increasing the number of processors and the problem size simultaneously???
  - Yes: ➡️ scalable parallel system
  - No: ➡️ non-scalable parallel system
- A scalable parallel system can always be made cost-optimal by adjusting the number of processors and the problem size.

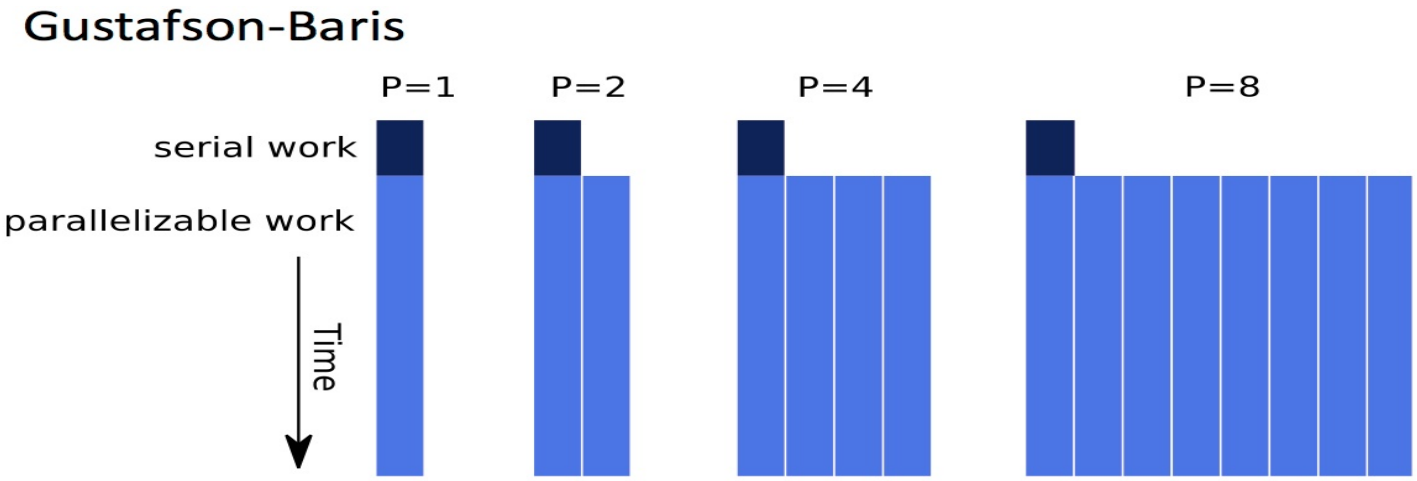
Interpreting Scalability Function



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* ($S_p = 1 + (p\text{-}1)f_{par}$)
  - 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





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?

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

Algorithmic Overhead

- All parallel algorithms are sequential when executed using one processor
- All parallel algorithms introduce 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!