Performance

Parallel Computing – COSC 3P93

Course Outline

- Introduction
- Parallel Hardware
- Parallel Software
- Performance
- Parallel Programming Models
- Patterns
- SMP with Threads and OpenMP
- Distributed-Memory Programming with MPI
- Algorithms
- Tools
- Parallel Program Design

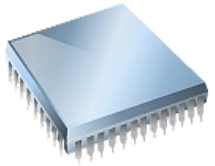
Today Class

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

What is 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

Why do we care about 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

> “The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”
>
> Charles Babbage, 1791 – 1871

What is Parallel Performance?

- Here we are concerned with performance issues when using a parallel computing environment
  - Performance with respect to parallel computation
- Performance is the *raison d'être* for parallelism
  - Parallel performance versus sequential performance
  - If the **performance** is not better
    - Parallelism is not necessary
- Parallel processing includes techniques and technologies necessary to compute in parallel
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, ...
- Parallelism must deliver performance
  - How? How well?

Performance Expectation (Loss)

- If each processor is rated at k MFLOPS and there are p processors, should we see k*p MFLOPS performance?
- If it takes 100 seconds on 1 processor
  - Shouldn't it take 10 seconds on 10 processors?
- Several causes affect performance
  - Each must be understood separately
  - But they interact with each other in complex ways
    - Solution to one problem may create another
    - One problem may mask another
- Scaling (system, problem size) can change conditions
- Need to understand performance space

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?

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
    - # 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

Scalability

- In general → a problem is **scalable**
  - If it can handle ever increasing problem sizes
- The problem is **strongly scalable**
  - If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size
- The problem is **weakly scalable**
  - If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads

Taking Timings

- What is time?
- Start to finish?
- A program segment of interest?
- CPU time?
- Wall clock time?

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a N-processor system
- S(p) (S_p) is the speedup

$$S(p) = \frac{T_1}{T_p}$$

- E(p) (E_p) is the efficiency

$$Efficiency = \frac{S_p}{N} = \frac{\left(\frac{T_{seq}}{T_{par}}\right)}{N} = \frac{T_{seq}}{N.T_{par}}$$

- Cost(p) (C_p) is the cost

$$Cost = N \times T_p$$

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

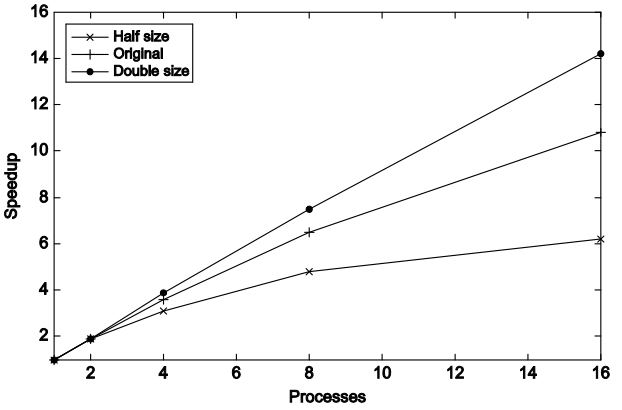
Speedups and Efficiency

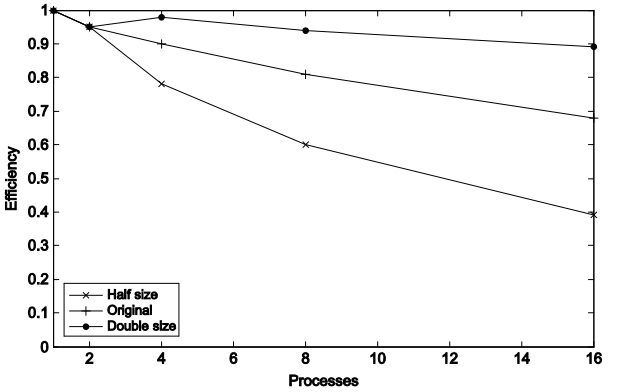
- Parallel program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
$E = S/p$	1.0	0.95	0.90	0.81	0.68

- Parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89





Speedups and Efficiency

- Parallel program on different problem sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Limits and Costs of Parallel Programming

- **Amdahl's Law**
  - Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized

$$speedup = \frac{1}{1 - P}$$

    - If none of the code can be parallelized
      - P = 0 and the speedup = 1 (no speedup)
    - If all of the code is parallelized
      - P = 1 and the speedup is infinite (in theory)
    - If 50% of the code can be parallelized
      - Maximum speedup = 2
        - The code will run twice as fast
    - Introducing the number of processors performing the parallel fraction of work
      - The relationship can be modelled by where P = parallel fraction, N = number of processors and S = serial fraction

Amdahl's Law (Fixed Size Speedup)

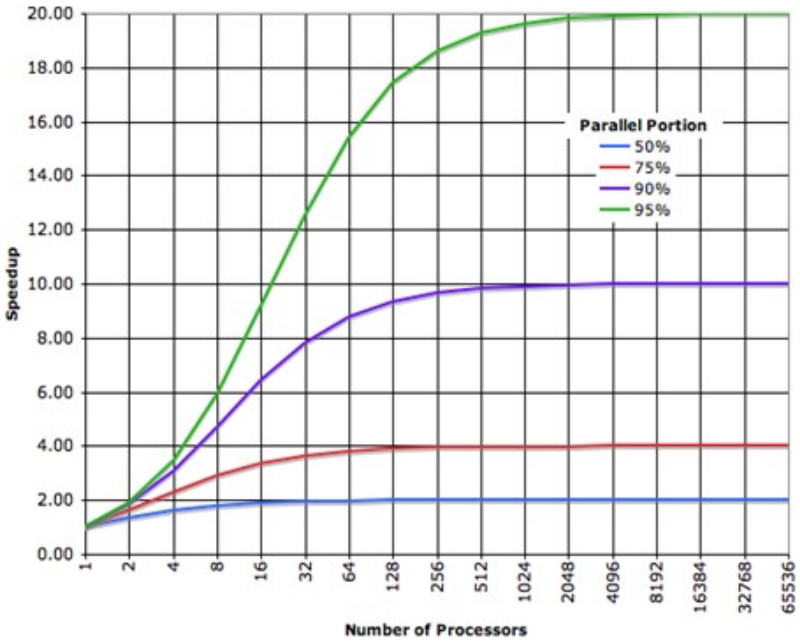
- Let S be the fraction of a program that is sequential
  - 1-S is the fraction that can be parallelized
- Let T1 be the execution time on 1 processor
- Let Tp be the execution time on N processors
- Sp is the speedup
  - Sp = T1 / Tp
  - = T1 / (S.T1 +(1-S).T1 / N))
  - = 1 / (S + (1-S) / N))
- As $N \rightarrow \infty$
  - Sp = 1 / S

Limits and Costs of Parallel Programming

- **Amdahl's Law**
  - Simplifying

$$Sp = \frac{1}{\frac{P}{N} + S}$$

  - where P = parallel fraction
  - N = number of processors
  - S = serial fraction



	Speedup			
N	**P=0.50**	**P=0.90**	**P=0.95**	**P=0.99**
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1000	1.99	9.91	19.62	90.99
10000	1.99	9.91	19.96	99.02
100000	1.99	9.99	19.99	99.90

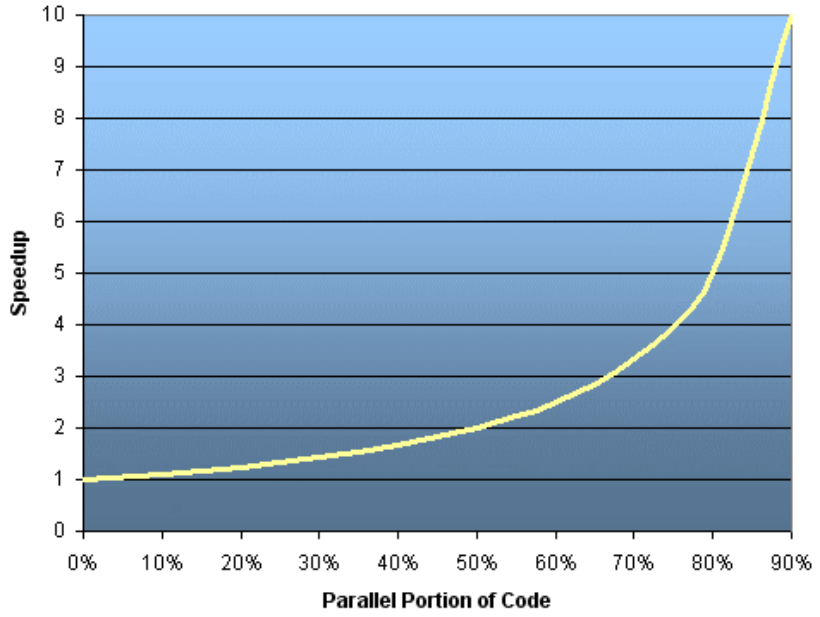
Limits and Costs of Parallel Programming

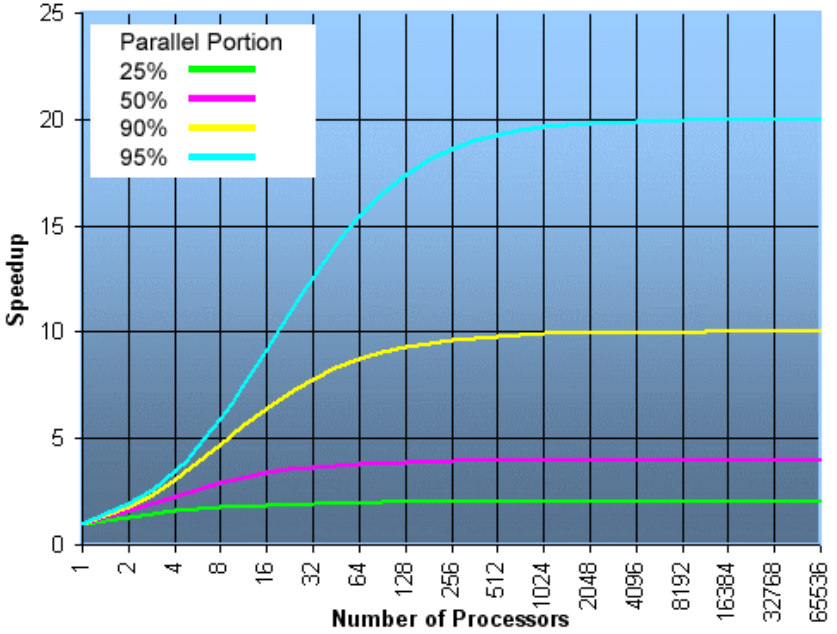
- **Amdahl's Law**
  - It soon becomes obvious that there are limits to the scalability of parallelism
  - For example

	Speedup			
N	**P=0.50**	**P=0.90**	**P=0.95**	**P=0.99**
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1000	1.99	9.91	19.62	90.99
10000	1.99	9.91	19.96	99.02
100000	1.99	9.99	19.99	99.90

Limits and Costs of Parallel Programming

- **Amdahl's Law**
  - It soon becomes obvious that there are limits to the scalability of parallelism
  - For example





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 ($N \to \infty$, $Sp \to S_\infty = 1 / S$)
  - 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
- ** Effect of overhead!
  - Assume a perfect, virtual "parallelization" of the serial code

$$T_{par} = T_{seq} / p + T_{overhead}$$

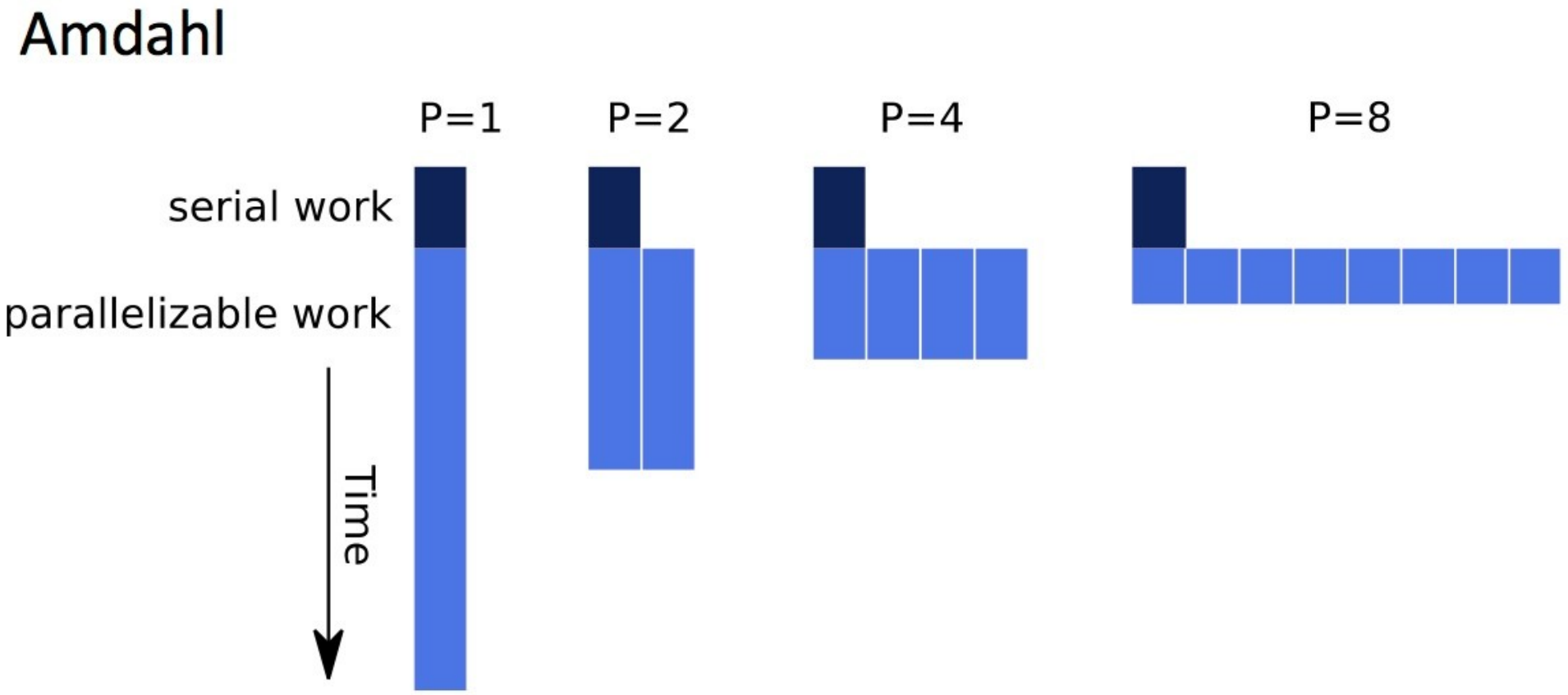
Gustafson-Barsis' Law (Scaled Speedup)

- Often interested in larger problems when scaling
  - How big of a problem can be run (HPC Linpack)
  - Constrain problem size by parallel time
- Assume parallel time is kept constant
  - Tp is constant
  - P_{seq} is the fraction of T_p spent in sequential execution
  - P_{par} is the fraction of T_p spent in parallel execution
- What is the execution time on one processor?
  - $T_s = P_{seq} . Tp + (1 – P_{seq}).N.Tp$
- What is the speedup in this case?
  - $S_p = T_s / T_p$
  - $= (P_{seq} . Tp + (1 – P_{seq}).N.Tp) / Tp$
  - $= P_{seq} + (1 – P_{seq}).N$
  - $= (1 - P_{par}) + P_{par}.N$

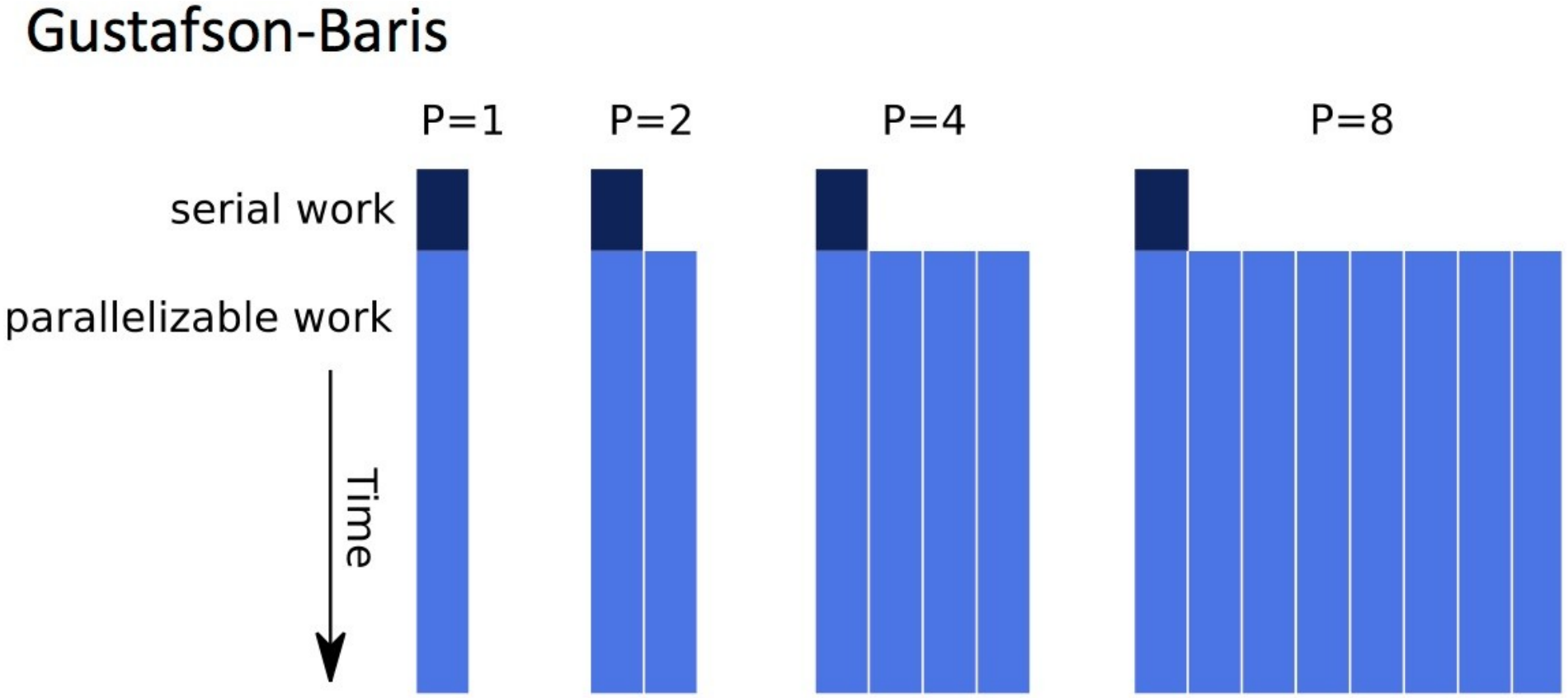
Gustafson-Barsis' Law and Scalability

- 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** → ($Sp = P_{seq} + NP_{par}$)
  - Speedup function includes the number of processors!!!
  - Can maintain or increase parallel efficiency as the problem scales

Amdahl versus Gustafson-Baris

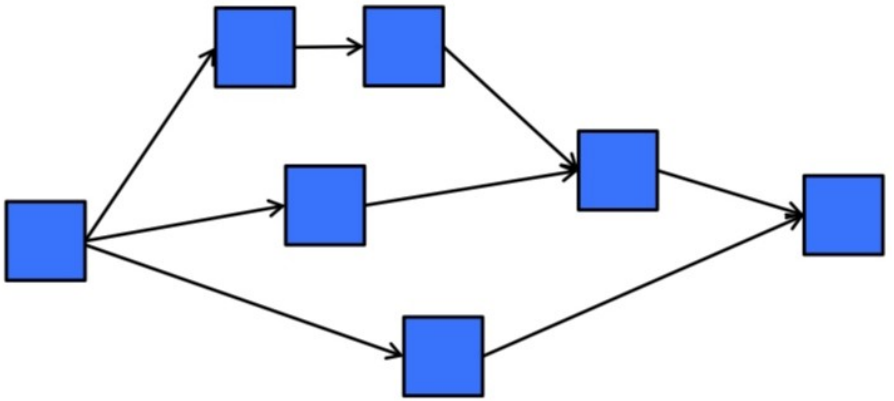


Amdahl versus Gustafson-Baris



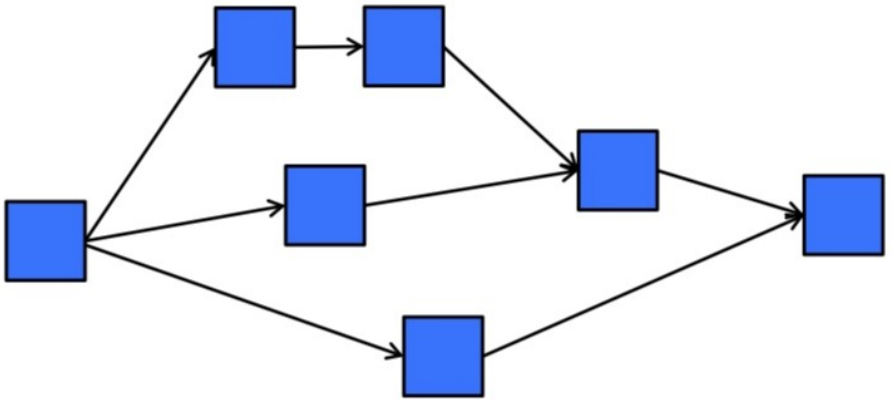
DAG Model of Computation

- 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



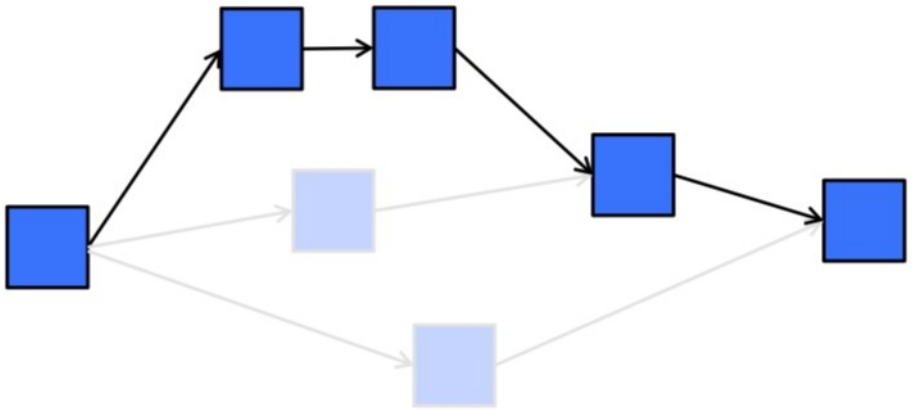
Work-Span Model

- T_P = 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_∞ = 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



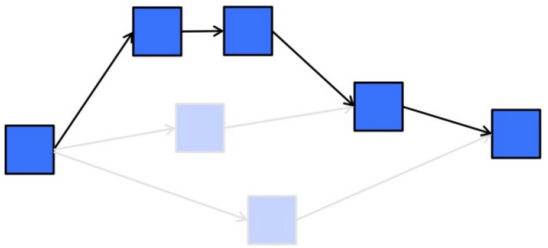
Work-Span Example

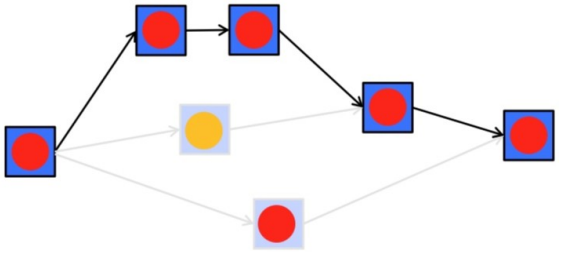
- Let each task take 1 unit of time
- DAG at the right has 7 tasks
- $T_1 = 7$
  - All tasks have to be executed
  - Tasks are executed in a serial order
  - Can the execution be in any order?
- $T_\infty = 5$
  - Time along the critical path
  - In this case, it is the longest pathlength of any task 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_∞ 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
  - Assume we can do so without overhead
  - $T_P \leq (T_1 - T_\infty) / P + T_\infty$



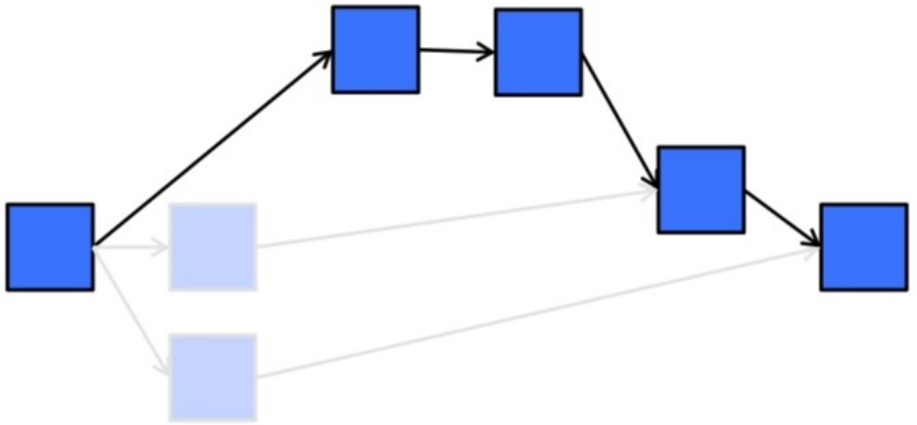


Consider Brent's Lemma for 2 Processors

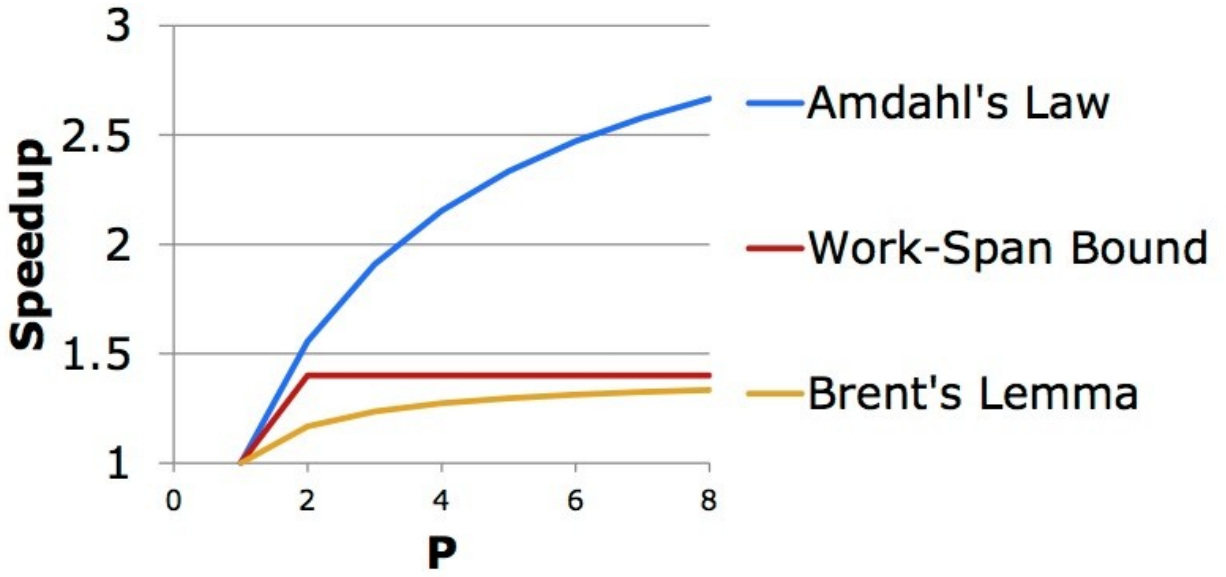
- T_1 $= 7$
- T_∞ $= 5$
- T_2 $\leq (T_1 - T_\infty) / P + T_\infty$

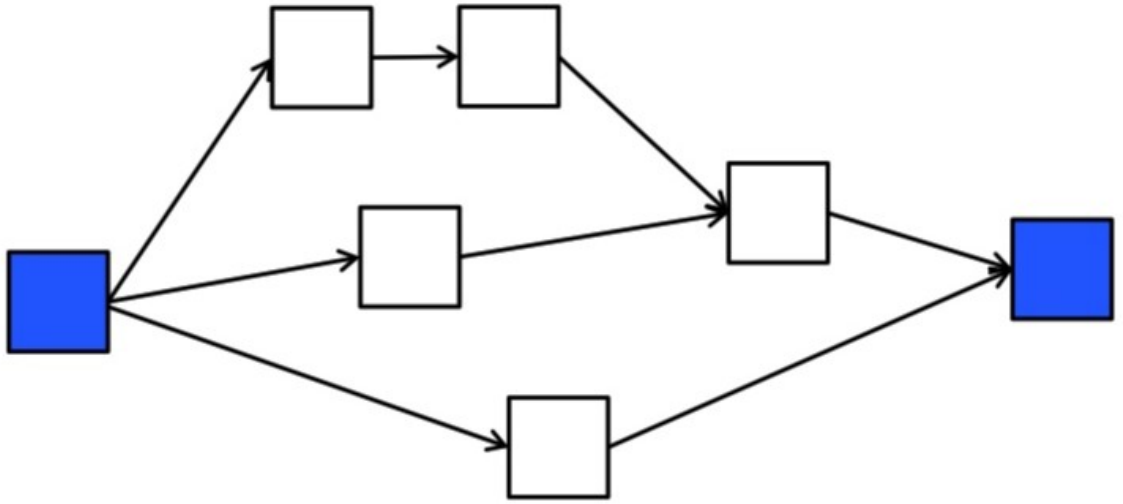
  $\leq (7 - 5) / 2 + 5$

  ≤ 6



Amdahl was an optimist!





Estimating Running Time

- Scalability requires that T_∞ be dominated by T_1

$$T_P \approx T_1 / P + T_\infty \text{ if } T_\infty << T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability
  - Even for finite P

Parallel Slack

- Sufficient parallelism implies linear speedup

$$T_P \approx T_1/P \quad \text{if} \quad T_1/T_\infty \gg P$$

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

Definitions for Asymptotic Notation

- Let T(N) mean the execution time of an algorithm
- Big O notation
  - T(N) is a member of O(f(N)) means that
    - $T(N) \leq c \cdot f(N)$ for constant c
- Big Omega notation
  - T(N) is a member of Ω(f(N)) means that
    - $T(N) \geq c \cdot f(N)$ for constant c
- Big Theta notation
  - T(N) is a member of Θ(f(N)) means that
    - $c_1 \cdot f(N) \leq T(N) < c_2 \cdot f(N)$ for constants c_1 and c_2

Limits and Costs of Parallel Programming

- **Quote**
  - You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!
- Certain problems demonstrate increased performance by increasing the problem size
    - 2D Grid Calculations 85 seconds 85%
    - Serial fraction 15 seconds 15%
  - We can increase the problem size by doubling the grid dimensions and halving the time step
    - This results in four times the number of grid points and twice the number of time steps
      - 2D Grid Calculations 680 seconds 97.84%
      - Serial fraction 15 seconds 2.16%
  - Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time

Limits and Costs of Parallel Programming

- **Complexity**
  - In general, parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude
    - Not only there are multiple instruction streams executing at the same time, but you also there are data flowing between them
  - The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle
    - Design
    - Coding
    - Debugging
    - Tuning
    - Maintenance
  - Adhering to **good** software development practices is essential when working with parallel applications
    - Especially if somebody besides you will have to work with the software

Limits and Costs of Parallel Programming

- **Portability**
  - Thanks to standardization in several APIs
    - MPI, POSIX threads, and OpenMP
  - Portability issues with parallel programs are not as serious as in years past → However…
  - The usual portability issues associated with serial programs apply to parallel programs
    - If you use vendor "enhancements" to Fortran, C, or C++
      - Portability will be a problem.
  - Standards exist for several APIs
    - However → implementations will differ in a number of details
      - Requiring code modifications in order to effect portability
  - Operating systems
    - They can play a key role in code portability issues
  - Hardware architectures
    - Characteristically highly variable and can affect portability

Limits and Costs of Parallel Programming

- **Resource Requirements**
  - The primary intent of parallel programming
    - To decrease execution wall clock time
      - However → more CPU time is required
    - For example → a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time
  - The amount of memory required can be greater for parallel codes than serial codes
    - The need to replicate data and for overheads associated with parallel support libraries and subsystems
  - Short running parallel programs
    - A decrease in performance compared to a similar serial implementation.
    - Overhead costs associated with setting up the parallel environment, task creation, communications, and task termination
      - A significant portion of the total execution time for short runs

Limits and Costs of Parallel Programming

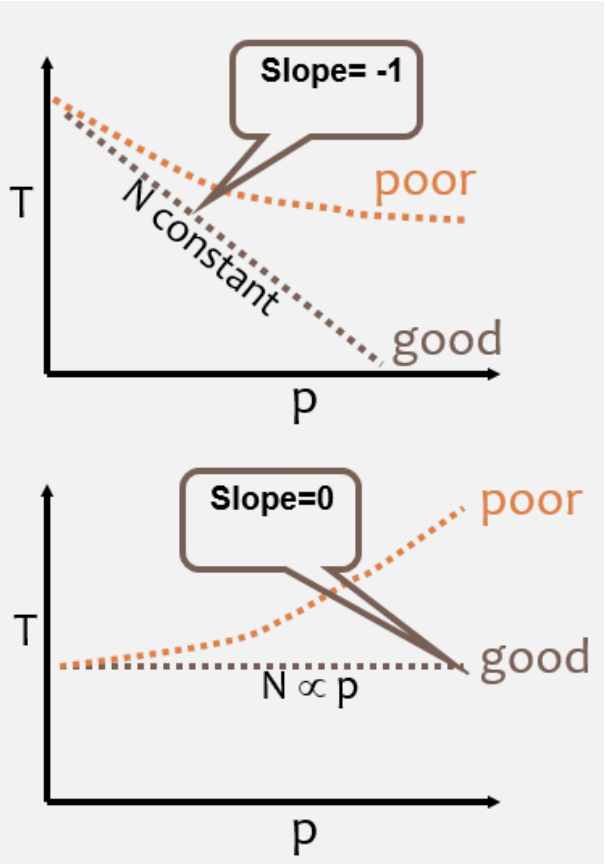
- **Scalability**
  - Two types of scaling based on time to solution: strong scaling and weak scaling
  - **Strong scaling**
    - The total problem size stays fixed as more processors are added
    - Goal is to run the same problem size faster
    - Perfect scaling means problem is solved in 1/P time (compared to serial)
  - **Weak scaling**
    - The problem size per processor stays fixed as more processors are added
      - The total problem size is proportional to the number of processors used.
    - Goal is to run larger problem in same amount of time
    - Perfect scaling means problem Px runs in same time as single processor run

Limits and Costs of Parallel Programming

- **Scalability**
  - Ability of a parallel program's performance to scale
    - Result of a number of interrelated factors
    - Simply adding more processors is rarely the answer
  - The algorithm may have inherent limits to scalability
    - Adding more resources causes performance to decrease
    - A common situation with many parallel applications
  - Hardware factors play a significant role in scalability
    - Memory-cpu bus bandwidth on an SMP machine
    - Communications network bandwidth
    - Amount of memory available on any given machine or set of machines
    - Processor clock speed
  - Parallel support libraries and subsystems software can limit scalability independent of your application

Limits and Costs of Parallel Programming

- **Scalability**



Class Recap

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

Next Class

- Performance
  - Scalable parallel execution
  - Parallel execution models
  - Isoefficiency