Amdahl's Law & Extensions

Speedup and Efficiency of Parallel Programs

- We need to predict the performance of a parallel program accurately (or as accurately as possible) before we try to implement.
- Analysis of the execution time exhibited by a parallel program helps us understand the barriers to higher performance and predict how much improvement can be realized by increasing the number of processors [cost benefit analysis]. To recap
  - Let p = number of processors (# PEs), n = size of the problem, $T_p(n)$ = execution time when p processors are available to work on a problem of size n [if n is omitted, we assume we are talking about the problem of same size); also that includes all overhead as well.
  - Assume [to make life simple]: (1) all processors are identical; (2) Ignore <u>all</u> overheads (communication, I/O, resource constraints, context switching, etc.) **Speed-up** = Sp = T_1/T_p, **Efficiency** = E(p) = S(p)/p [normalized speed-up] ; T_i is the parallel execution time with i processors.
  - Speed-up is linear if efficiency E(p) = α, some constant ≤ 1 and is ideal if $\alpha = 1$. Linear speed-up is not possible for an algorithm for all possible problem sizes, because of (1) communication overhead (2) resource contentions (3) structure of software.
  - The operations performed by parallel algorithm can be put into three categories:
    - Computations that must be performed sequentially
    - Computations that can be performed in parallel
    - Parallel overhead (communication operations and redundant computations)

Amdahl's Law[*]

Amdahl's law states that if a portion (fraction) of a computational task, **f** , can be improved by a factor p, and the other portion (1 – f) [inherently sequential fraction] cannot be parallelized, then the portion that cannot be parallelized will quickly dominate the performance, and further improvement of the improvable [parallelizable] portion will have little effect.

$$S(n,p) \le \frac{1}{1-f+\frac{f}{p}} \qquad \lim_{p\to\infty} S(n,p)_{Amdahl} = \frac{1}{1-f}$$

Note: ***p*** is a single parameter characterization of hardware (to speed up the parallelizable fraction) [p is not necessarily the number of individual Processing Elements] and S(n, p) denotes the overall speed-up of the computation. Implicit assumptions are:

- The parallelizable part of the program is infinitely parallelizable [Never true!] without scheduling or communication overhead and the problem size [workload] is constant [Never true].
- The speedup is a measure of the time reduction of a problem of a given instance size. Amdahl's law is thus also called the fixed-size speedup model. Note that the simplistic formula does not explicitly involve size of the problem.

- When f is small, optimizations will have little effect.
- As p approaches infinity, speedup is bounded by 1/(1 – f).
- **Note:** Amdahl's law completely disregards the problem size as well as parallel overhead; in its original formulation, the problem size was mentioned at all.

[*]Gene M. Amdahl, "*Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*", Proc. Am. Federation of Information Processing Societies Conf., AFIPS Press, 1967, pp. 483-485.

[#] "Everyone knows Amdahl's Law, but quickly forgets it" – Thomas Puzak, IBM, 2007

Examples

- Consider a program where 90% of it can be executed 10 times faster $\Rightarrow$ (1 – f) = 0.1 and n = 10 and overall speed-up is 1/(0.1 + 0.9/10) = 5.26
- Consider a program where 80% of it can run 20% faster $\Rightarrow$ f = 0.8 and n = 1.2 and overall speed-up is 1/(0.2 + 0.8/1.2) = 1.153
- You have a system that contains a special processor for floating-point operations – suppose 60% of your computations can use that processor and when the program uses that special processor, the speed-up of the processor is 40% faster than when it doesn't use it $\Rightarrow$ f = 0.6, n = 1.4 and overall speed-up is 1.206. Now, in order to improve the speed-up you are considering 2 options:
  - Option 1: Modifying the compiler so that 70% of the computations can use the floating point processor. Cost of this option is $50K
  - Option 2: Buying a new floating point processor. The speed-up of the new processor is 100% faster than when it doesn't use it. Say, 50% of the computations can use the processor and the cost is $60K.

Question: Which option to recommend? Why?

- Option 1: f = 0.7, n = 1.4, Overall Speed-up = 1/(0.3 + 0.7/1.4) = 1.25. So, Cost/Speed-up = $50K/1.25 = $40K.
- Option 2: f =0.5, n = 2, Overall Speed-up = 1/(0.5 +0.5/2) = 1.33, So, Cost/Speed-up = $60K/1.33 = $45.1K

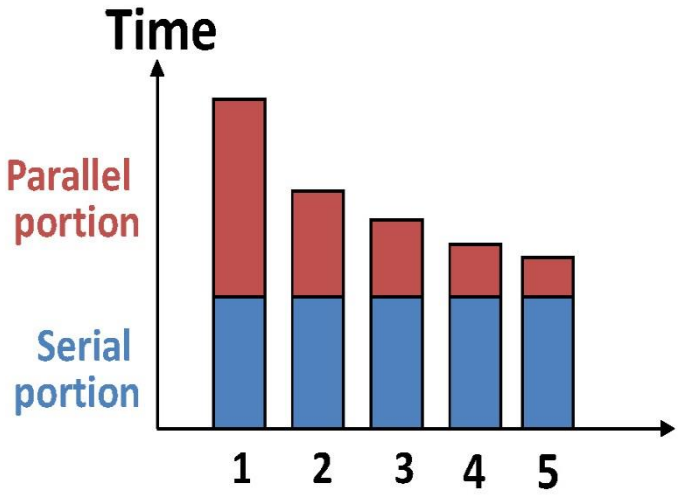
Examples [Contd.]

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum)
  - 10 PEs: f = 100/110 = 0.909; speedup = 1/(0.91+.909/10) = 5.5
  - 100 PEs: f ≈ 0.909; speedup = 1/(0.91+ .909/100) = 10
- Consider summing 10 scalar variables and two 100 by 100 matrices (matrix sum)
  - 10 PEs: f = 10000/10010 =.999; speedup = 9.9
  - 100 PEs: speedup is 91
- Consider a system that contains a special processor for floating-point operations whose speed-up is 15. We find that 50% of the computations can use the floating point processor.
  - So, $f = 0.5$, $n = 15$, Overall Speed-up = 1/(0.5 + 0.5/15) = 1.876
  - Suppose the compiler is modified so that 75% of the computations can use the processor $\Rightarrow f = 0.75$, $n = 15$, Overall Speed-up = 1/(0.25 + 0.75/15) = 3.33.
  - What fraction of the computations should be able to use the floating point processor to achieve a overall speed-up of 2.25? $\Rightarrow f = ??$, $n = 15$, Overall Speed-up = 2.25 = $1/(1 - f + f/15)) \Rightarrow f = 0.495$
- What other kinds of examples can we imagine? Try yourself ; if you didn't at least a few, you didn't get it.

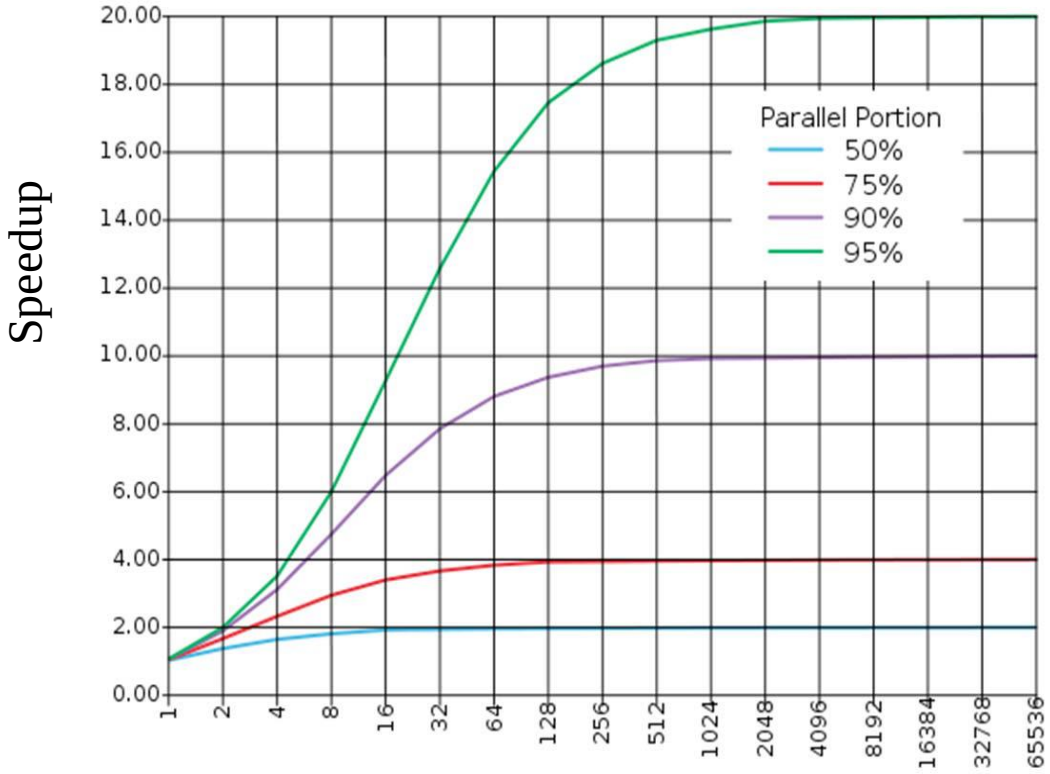
More Practice Problems

- Suppose you work as a truck driver, and you have been hired to carry a load of potatoes from Boise, Idaho, to Minneapolis, Minnesota, a total distance of 2500 kilometers. You estimate you can average 100 km/hr driving within the speed limits, requiring a total of 25 hours for the trip.
  - You hear on the news that Montana has just abolished its speed limit, which constitutes 1500 km of the trip. Your truck can travel at 150 km/hr. What will be your speedup for the trip?
  - You can buy a new turbocharger for your truck at www.fasttrucks.com. They stock a variety of models, but the faster you want to go, the more it will cost. How fast must you travel through Montana to get an overall speedup for your trip of 5/3?
- The marketing department at your company has promised your customers that the next software release will show a 2× performance improvement. You have been assigned the task of delivering on that promise. You have determined that only 80% of the system can be improved. How much would you need to improve this part to meet the overall performance target?

Speedup achieved through parallelism is limited by the non-parallel portion of the program



Number of Processors



Number of Processors

Amdahl's Law assumes that we are trying to solve a problem of fixed size as quickly as possible

Better Formulation

- Assume n is the size of a given instance of a problem and p is the number of processors (identical); then, assume that inherently sequential computations is σ(n), potentially parallel computations is φ(n), and (inter-processor) parallel overhead is μ(n, p). Then, speedup is a function of both n and p; we get

$$S(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\mu(n,p)}, \quad E(n,p) = \frac{S(n,p)}{p}$$

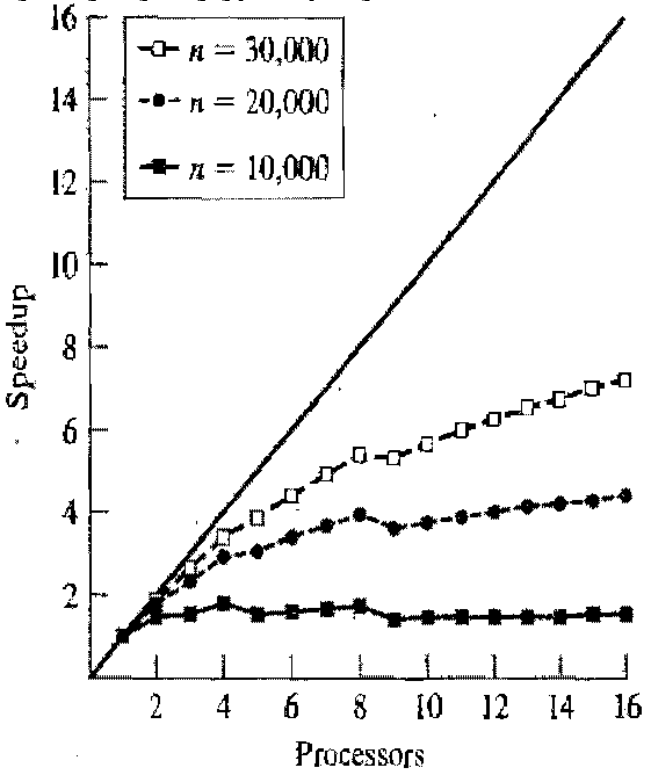
  Note: We made the optimistic assumption that the parallel portion of the computation can be divided perfectly among the processors; if not, the speedup bound would be somewhat (sometimes drastically) less.

- **Note**: Assume for now, μ(n, p) = 0, for simplicity; our original f, the parallelizable portion of the computation = φ(n)/(σ(n)+φ(n)) ⇒ speedup S(n, p) = 1/{1 – f + f/p)}; we get back our original Amdahl's Law!! [Note: 0 ≤ f ≤ 1] Amdahl's Law is based on the assumption that we are trying to solve a problem of fixed size as quickly as possible.
- Typically, μ(n,p) has lower complexity than φ(n); when this not true, Amdahl's law overestimates expected speed-up.
- Example: Next page.

Example

- Consider a parallel version of a sequential program with time complexity $\Theta(n^2)$, where n is the size of the dataset Assume the time needed to input the dataset and output the result is (18000 + n) }μsec – this constitutes the sequential portion of the program. The computational portion of the program can be executed in parallel; it has execution time (n^2/100) μsec. By Amdahl's Law [bound]

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(28{,}000 + 1{,}000{,}000/p)\ \mu\text{sec}}$$



- Assume now the parallel version has ⌈log n⌉ communication points; communication time at each point is 10000⌈log p⌉ + (n/10) μsec [μ(n, p) element]. The prediction for the speedup achievable by the parallel program solving a problem of size

$$\psi \leq \frac{(28{,}000 + 1{,}000{,}000)\ \mu\text{sec}}{(42{,}000 + 1{,}000{,}000/p + 140{,}000\lceil \log p \rceil)\ \mu\text{sec}}$$

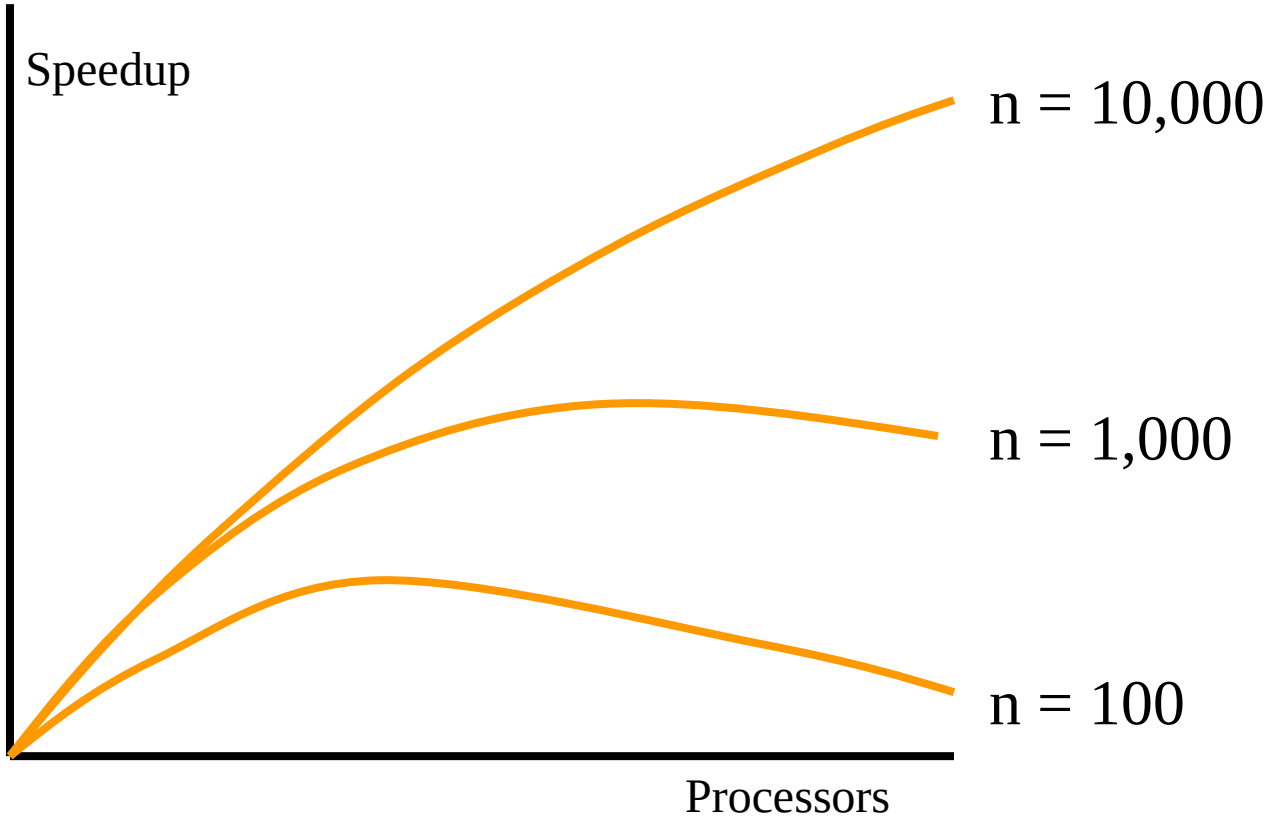
- Since μ(n, p) < φ(n), increasing tile size of the problem increases the computation time faster than it increases the communication time. Hence for a fixed number of processors. speedup is usually an increasing function of the problem size

Practice Problems

- Consider a problem of adding n integers using p processors where the communication time between two processors is (a) 10, (b)100 and the time to add two integers is (a) 1, (b) 10. What can we say about the system from what we have seen so far?
- Consider multiplying 2 n × n matrices with the above assumptions (assume the time to multiply two integers is (a) 2, (b) 10 (c) 30. Redo the exercise and Draw clear and precise conclusions with graphs and succinct reasoning [Write a program to draw a meaningful chart]
- We can generalize Amdahl's law to the case when multiple enhancements are possible. Three enhancements with the following speedups are proposed for a new architecture: $Speedup_1$ = 30, $Speedup_2$ = 20, $Speedup_3$ = 15. Only one enhancement is usable at a time (but multiple can be used over the entire application). If enhancements 1 and 2 are each usable for 25% of the time, what fraction of the time must enhancement 3 be used to achieve an *overall speedup* of 10 for the entire application?
- Amdahl's Law can be generalized to handle multiple enhancements. If only one enhancement can be used at a time during program execution, then Speedup $= 1/\{1 - \Sigma_i FE_i + \Sigma_i FE_i/SE_i\}$, where FE_i is the fraction of time that enhancement i can be used and SE_i is the speedup of enhancement i.
- For a single enhancement the equation reduces to the familiar form of Amdahl's Law. With three enhancements we have Speedup= $1/\{1 - (FE_1 + FE_2 + FE_3) + (FE_1/SE_1) + (FE_2/SE_2) + (FE_3/SE_3)\}$. Substituting in the known quantities gives Speedup= $1/\{1 - (0.25 + 0.25 + FE_3) + (0.25/30) + (0.25/20) + (FE_3/15)\}$. Solving the above equation for FE_3 gives FE_3= 0.45. Thus, the third enhancement must be usable 45% of the time.
- Try different kinds of variations of variations; you can draw conclusions.

Limitations of Amdahl's Law

- Limitations of Amdahl's Law: Ignores μ(n,p); overestimates speedup achievable; treats problem size as a constant; shows how execution time decreases as number of processors increases; could also be overly pessimistic.
- **Note:** Typically μ(n,p) has lower complexity than φ(n)/p; as n increases, φ(n)/p dominates μ(n,p); as n increases, speedups potentially increase; let's see an example.



Scaled (Fixed size) Speed-up

- A tacit assumption in Amdahl's law is that the problem size, or the workload, is fixed to that which runs on the unenhanced system. The speedup emphasizes time reduction of a given problem. Amdahl's law is thus also called the fixed-size speedup model. In 1988, Gustafson introduced the concept of scalable computing and the fixed-time speedup model. The fixed-time speedup model argues that powerful machines are designed for large problems and problem size should (and do) scale up with the increasing of computing capability. For many practical workloads (e.g., real time applications), the problem size scale-up is bounded by the execution time.
- The approach is as follows: Consider an arbitrary workload w; it takes certain amount of time, say t, using a single processor; consider the same problem with a larger size of workload w_1 (assuming the scale up in load is in the parallelizable part only); assume it takes the same time t to execute on a p processors; the speed up obtained in the second case is called the fixed time speedup.
- References:
  - Xian-He Sun, Yong Chen, "Reevaluating Amdahl's law in the multicore era", J. Parallel Distrib. Comput. 70 (2010) 183 – 188.
  - Mark D. Hill, Michael R. Marty, "Amdahl's Law in the Multicore Era", IEEE Computer Magazine, July 2008, pp. 33 – 38.
  - Dong Hyuk Woo and Hsien-Hsin S. Leepp, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era", IEEE Computer Magazine, December 2008, 24 – 31.

Fixed Time Speedup

- The fixed-time speedup model [also known as Gustafson-Barsis's Law] argues that powerful machines are designed for large problems and problem size should scale up with the increasing of computing capability. For many practical workloads (e.g. real time applications), the problem size scale-up is bounded by the execution time. Thus, the fixed-time speedup is defined as:

$$Speedup_{FT} = \frac{Sequential\ Time\ of\ Solving\ Scaled\ Workload}{Parallel\ Time\ of\ Solving\ Scaled\ Workload}$$

- Assume the original workload w, and the scaled workload w_1, finish in the same amount of time with sequential processing and parallel processing with p processors, respectively; and that the scale of the workload is in the parallel processing part only; we have $w_1 = (1 - f)w + fpw$ [f is the fraction of time spent in the parallel computation performing parallel operations]. Therefore,

$$Speedup_{FT} = \frac{\text{Sequential Time of Solving } w_1}{\text{Parallel Time of Solving } w_1} = \frac{\text{Sequential Time of Solving } w_1}{\text{Sequential Time of Solving } w}$$

$$\leq \frac{w_1}{w} = \frac{(1-f)w + fpw}{w} = (1-f) + pf = p - (1-f)(p-1)$$

- It states that the fixed-time speedup is a linear function of m if the workload is scaled up to maintain a fixed execution time. Thus, this suggests that it is beneficial to build a large-scale parallel system as the speedup can grow linearly with the system size. [Note: Many applications cannot scale up to meet the time bound constraint due to some physical constraints, e.g. memory limitation]

*J.L. Gustafson, Reevaluating Amdahl's Law, Communications of the ACM (1988).

Simple Examples

- An application running on 10 processors spends 3% of its time in serial code. What is the scaled speedup of the application? [Note: f = 0.97, m =10 and hence the scaled speedup is 9.73]
- 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? [m = 8, then $7 = (1 - f) + 8f \Rightarrow f = 6/7 \Rightarrow$ serial part is 1/7 or $\approx 14\%$].
- An application executing on 64 processors using 5% of the total time on non-parallelizable computations. What is the scaled speedup? f= 0.05, so speedup is $\leq$ 60.85.
- Remember the alternative formulation, and note that fixed time speed up also

$$S(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/\ p+\mu(n,p)}, \quad E(n,p) = \frac{S(n,p)}{p}$$

ignores μ(n,p) as is done in Amdahl's law. Thus we still get a overestimate of speedup. **Question:** What can we say about μ(n,p)? Do or can we know how does that function behave?

Karp-Flatt Metric[*]

- The Karp-Flatt metric can help you decide whether the principal barrier to speedup is the amount of inherently sequential code or parallel overhead.
- In many cases, assuming a single processor is only p times slower than p processors is overly optimistic. For example, imagine solving a problem on a parallel computer with 16 processors, each with one GB of local memory. Suppose the dataset occupies 15 GB, and the aggregate memory of the parallel computer is barely large enough to hold the dataset and multiple copies of the program. If we tried to solve the same problem on a single processor, the entire dataset would not fit in primary memory. If the working set of the executing program exceeded one GB, it would begin to thrash (page faults), taking much more than 16 times as long to execute the parallel portion of the program as the group of 16 processors.
- Example: An application executing on 64 processors requires 220 seconds to run. Benchmarking reveals that 5 percent of the time is spent executing serial portions of the computation on a single processor. What is the scaled speedup of the application?
- Both Amdahl's Law and Gustafson-Barsis's Law ignore $\mu(n,p)$, the parallel overhead term and hence overestimate speedup.
- Recall that we have represented the execution time of a parallel program executing on p processors as $T(n, p) = \sigma(n) + \varphi(n) + \mu(n,p)$ where $\sigma(n)$ is the inherently sequential component, $\varphi(n)$ is the portion of parallelizable computation and $\mu(n,p)$ is the parallel overhead (processor communication and synchronization, and redundant computations, etc.)
- When $p = 1$, $T(n, 1) = \sigma(n) + \varphi(n)$ [no parallel overhead]
- Define the experimentally determined serial fraction **e** of the parallel computation to be

$$\mathbf{e = (\sigma(n) + \mu(n,p))/T(n,1)}$$

[*]Alan H. Karp and Horace P. Flatt, "Measuring Parallel Processor Performance", CACM, Vol. 33(5), 1990, pp. 539 – 543.

Karp-Flatt Metric

- Rewrite the parallel execution time as

  $T(n, p) = T(n, 1)e + T(n, 1)(1 – e)/p$ [Assuming $p > 1$]

- Since speedup $S(n, p) = T(n, 1)/T(n, p) = \psi$ (say), we have $T(n, 1) = T(n, p)\,\psi$ or

  $T(n, p) = T(n, p)\,\psi e + T(n, p)\,\psi(1 – e)/p \Rightarrow 1 = \psi e + \psi(1 – e)/p$

$$\frac{1}{\psi} = e + (1-e)/p \Rightarrow\Rightarrow e = \frac{1/\psi - 1/p}{1-1/p}$$

- The experimentally determined serial fraction is a useful metric for two reasons.
  - It takes into account the parallel overhead $\mu(n, p)$ term.
  - It accounts for other sources of overhead or inefficiency that are ignored in our simple model of parallel execution time. For example, we assume that p processors execute the parallelizable portion of the computation p times as quickly as a single processor. [That is why the $\varphi(n)$ term in T (n, 1) becomes the $\varphi(n)/p$ term in T(n, p).] For example, suppose we have 19 equal and undividable pieces of work, each of which takes one unit of time to complete. If six processors are available, one processor must take four pieces while the other processes take three. The parallel execution time is 4, not 19/6.
  - By using the experimentally determined serial function, we can determine whether this efficiency decrease is due to (1) limited opportunities for parallelism or (2) increases in algorithmic or architectural overhead.

Karp-Flatt Metric

- Example 1: Benchmarking some parallel program on 1, 2, ... , 8 processors produces the following speedup results: What is the primary reason for the parallel program achieving a speedup of only 4.7 on 8 processors?

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71

Compute ψ and e:

p	2	3	4	5	6	7	8
ψ	1.82	2.50	3.08	3.57	4.00	4.38	4.71
e	0.10	0.10	0.10	0.10	0.10	0.10	0.10

Since the experimentally determined serial fraction is not increasing with the number of processors, the primary reason for the poor speedup is the limited opportunity for parallelism, i.e., the large fraction of the computation that is inherently sequential.

- Example 2: Consider another similar example:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71

Compute ψ and e:

p	2	3	4	5	6	7	8
ψ	1.87	2.61	3.23	3.73	4.14	4.46	4.71
e	0.070	0.075	0.080	0.085	0.090	0.095	0.1

Since the experimentally determined serial fraction is steadily increasing as the number of processors increases, the principal reason for the poor speedup is parallel overhead. This could be time spent in process startup, communication, or synchronization, or it could be an architectural constraint.

Memory Bound Speedup

- Consider the physical constraint of memory limitations of a single processor. Let w* be the scaled workload under a memory space constraint. Assume that each computing node is a processor-memory pair. Increasing the number of processors, then, will increase the memory capacity as well. Let **y = g(x)** be the function that reflects the parallel workload increase factor as the memory capacity increases **m** times. That is, w = g(M), and w* = g(m·M), where M is the memory capacity of one node. We have w* = g(m·g^{-1}(w)). Thus memory-bounded speedup is:

$$Speedup_{MB} = \frac{\text{Sequential Time of Solving w*}}{\text{Parallel Time of Solving w*}} = \frac{(1-f)w + f \cdot g(m \cdot g^{-1}(w))}{(1-f)w + \frac{f \cdot g(m \cdot g^{-1}(w))}{m}}$$

- Consider a function* g(x) = ax^b, a and b are two rational numbers; then, g(m, x) = $a(mx)^b$ = $m^b \cdot ax^b$ = m^bg(x) = α(m)g(x), where α(m) is the power function with a = 1. Considering only the highest degree term to represent the complexity of the function, we get

$$Speedup_{MB} = \frac{(1-f)w + f\alpha(m)w}{(1-f)w + \frac{f \cdot \alpha(m)w}{m}} = \frac{(1-f) + f \cdot \alpha(m)}{(1-f) + \frac{f \cdot \alpha(m)}{m}}$$

*Most algorithms have a polynomial complexity in terms of computation and memory requirement,

An Example: compute α(x) for matrix Multiplication

- The computation requirement of matrix multiplication is $x = 2N^2$ and the memory requirement is $x = 3N^2$, where N is the dimension of the two $N \times N$ source matrices. Thus:

$$g(x) = 2\left(\frac{x}{3}\right)^{3/2} = \frac{2}{3^{3/2}} x^{3/2}, \text{ and } \alpha(x) = x^{3/2};$$

$$Speedup_{MB} = \frac{(1-f) + f\alpha(m)}{(1-f) + \frac{f\alpha(m)}{m}} = \frac{(1-f) + f \cdot m^{3/2}}{(1-f) + f \cdot m^{1/2}}$$

- In general, if we assume each element stored in memory will be used at least once, we have $w^* \geq w_1$, and the memory-bounded speedup is greater than or equal to the fixed-time speedup.
- This is a generalization of $Speedup_{Amdahl}$ and $Speedup_{FT}$; note that we get the first case with $\alpha(m) = 1$ and the second case with $\alpha(m) = m$.
- In general, the computational workload increases faster than the memory requirement, thus $\alpha(m) \geq m$ and the memory-bounded speedup model gives a higher speedup than the fixed-size and fixed-time speedup. Memory-bounded speedup seems natural for domain decomposition-based applications and can be applied at different levels of a memory hierarchy system. It becomes more and more important with increasing awareness of the memory-wall problem.

A Simplistic Model of Parallel Software (Algorithm)

- Speed-up vs. Efficiency
- *Read the Lazoska Paper

Motivation

- "Implement and Measure" is expensive and time consuming.
- Abstract Modeling: Use minimum info, gain insight into the problem, answer fundamental questions; select the promising ones. Consider granularity of subtasks.

Bounds expressed in terms of one or a few parameters are desirable than precise solutions that require complete information about the software and the hardware.

- Complete info is often not available [volume is often prohibitively large]; for most software, parallelism also depends on size of problem.
- Bounds based on simple characterizations often yield interesting, practical, and more insightful knowledge.

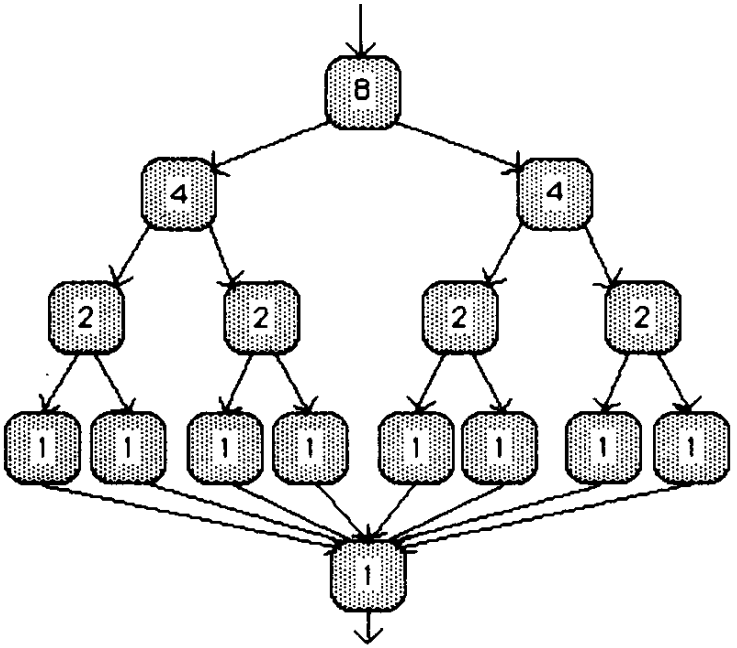
Consider Amdahl's Law [we want to get something similar]

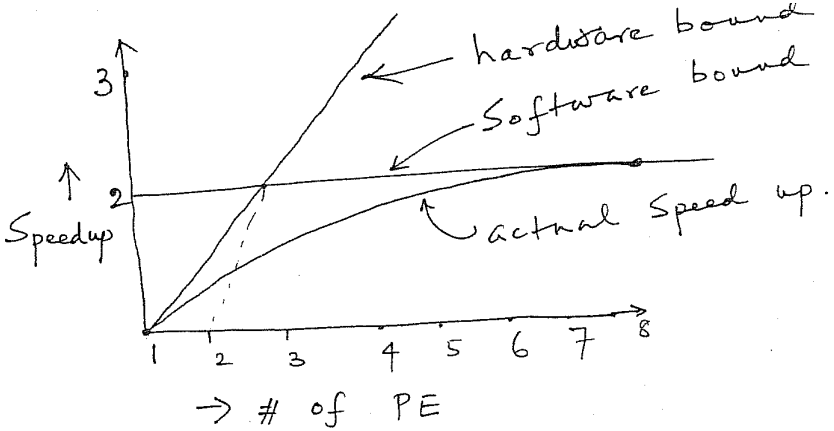
A Simple System Model

- Traditional precedence graph (acyclic directed graph); weights in the vertices indicate execution times; a directed edge from node ‘a’ to node ‘b’ indicates that ‘a’ precedes ‘b’ or ‘b’ cannot start until ‘a’ is completed.
- n identical processors (unit speed)
- Ignore all overhead like memory contention, communication delay, semaphores, any other resource constraints (they are important, outside the scope of this simple model)
- Work-conserving scheduling discipline – never idle a free processor when there is a ready task
- One possible scheduling scheme could be a simple rule: complete the sub tasks level by level – not optimal by any means.

Simple Example

- We assume tasks cannot be preempted
- Total Service = T_1 = 33
- Longest path length = 16
- T_2 = 8 + 4 +4 +4 +1 =21
- T_3 = 8 + 4 + 2 + 2 + 1 +1 + 1 =18
- T_4 = 8 + 4 + 2 + 2 + 1 =17
- T_8 = 8+4 +2 +1 +1 = 16
- Any T_i, i > 8, remains 16
- Speed-up is really bounded by software (algorithm)
- How do we characterize inherent parallelism in the software?
- Assume processor sharing – how does the computation change?





Consider Processor Sharing

- A ***work-conserving discipline*** is one that never leaves idle a subtask that is eligible for execution when there is a processor available.
- ***Processor Sharing discipline***: if k subtasks are eligible for execution and there are n available processors (n < k), each subtask receives service at a rate that is n/k times the rate at which it would receive service if a processor were dedicated to it. Note that this is doable using preemption at no cost and the overall scheduling becomes easier; otherwise, optimal scheduling is NP –hard. Another advantage is that any processor idles only when forced by the precedence constraints and/or number of processors used in the system.
- If we redo the previous example using processor sharing, T_1, T_2 remain the same while T_3 becomes 8+4+8/3+8/3 ≈17.2; re-compute all other values.

Average Parallelism Measure (A)

We can define A in 4 different but equivalent ways:

- With unbounded parallelism, A is the average number of PEs that are busy over the entire execution time
- A is the speed-up $S_\infty = T_1/T_\infty$
- A is the ratio of total service requirement and length of the longest path
- A is the intersection point of hardware and software bound.

Observations:

- Hardware bound on speed-up is imposed by the number of processors; that can be achieved only when all the processors can be kept busy.
- Software bound is imposed by the structure of the software; speed-up can never be bigger than A.
- If an unbounded number of processors is available, the execution time of a software system is simply the total service demand along some longest path.
- The intersection point of the hardware and software bounds on speedup is significant: when additional processors are allocated, it is certain that there is not enough parallelism in the software to keep all of the processors busy all of the time. This intersection point is the point where n (the hardware bound) is identical to the ratio of the total service demand to the length of the longest path (the software bound).

Theorem: $S(n) \geq \frac{nA}{n+A-1}$ $\left[\text{or } E(n) \geq \frac{A}{n+A-1}\right]$

If T_∞ is the execution time with unbounded parallelism, total workload is $T_1 = T_\infty.A$. Let I_n = total idle time on all PEs when the software runs on n PEs. So,

$$T_n = \frac{1}{n}\{I_n + T_\infty.A\} \text{ or } S(n) = \frac{nA}{A + I_n/T_\infty}$$

The worst scenario for idle time is that all the time all but one PE are idle [work conserving principle] and the precedence graph is a line graph] $\Rightarrow I_n \leq T_\infty(n-1)$. Substitution yields the theorem.

Note: This is a lower bound on S(n). Also, if $n << A$, $S(n) \rightarrow n$ and if $n >> A$, then $S(n) \rightarrow A$. It can be shown that the lower bound is indeed tight.

The Upper bound is Tight

- We show that the speedup bound can be attained by means of an example. Consider first-come-first-served scheduling, and a software system that consists of a subtask that, upon completion, enables kn + 1 additional subtasks, where kn is some multiple of n that is greater than A. If we constrain the service times of each of the kn + 1 additional subtasks to be identical, the service time of the first subtask can be derived as T,(kn + 1 - A)/kn, and that of each of the remaining subtasks as T,(A - l)/kn. The speedup is then given by

$$S(n) = \frac{(kn+1)\,T_\infty \dfrac{A-1}{kn} + T_\infty \dfrac{kn+1-A}{kn}}{T_\infty \dfrac{kn+1-A}{kn} + (k+1)\,T_\infty \dfrac{A-1}{kn}}.$$

Speed-up vs. Efficiency

For any n, any software structure and any work preserving discipline, it can be shown that

$$E(n) + \frac{S(n)}{A} > 1$$

where E(n) is attained efficiency and S(n)/A is the attained fraction of maximum possible speed-up.

- By adding processors, speed up can be purchased at the cost of efficiency.
- Very poor anomalous situations (low speed up, low efficiency) cannot exist; parallel processing is inherently robust

Efficiency Penalty for a given speed-up

Suppose, appropriate number of processors are employed to get some speed up (no greater than A, of course!!). How much do we pay in terms of loss of efficiency? Observe

$$S(n) > \frac{nA}{n+A-1} \quad \Rightarrow n < \frac{S(n)(A-1)}{A-S(n)}$$

$$\text{So, } \frac{S(n)}{n} = E(n) \geq \frac{A-S(n)}{A-1}$$

This is good when A > 1 [strictly sequential programs wouldn't work]. For low speed up, loss in efficiency is low; but for high speed up efficiency may be arbitrarily low.

Average Parallelism & Trade-off

Speed up is upper bounded by hardware (n) and software (A) as well as lower bounded by nA/n+A-1. Suppose we make an estimate as

$$\mathbb{S}(n) = 2 \times \frac{\min(n, A) \times \frac{nA}{n + A - 1}}{\min(n, A) + \frac{nA}{n + A - 1}}$$

This estimate has always a relative error less than 34% i.e.,

$$\frac{|\mathbb{S}(n) - S(n)|}{S(n)} < 0.34$$

The proof involves simple algebra [see the paper]. Thus, an estimate of speed up based on average parallelism alone is guaranteed to have less than 34% error.; additional knowledge can provide only a modest or marginal benefit.

Effect of Sequential Thread

Let f be the fraction of work that is inherently sequential; then $0 \leq f \leq 1$ and also $0 \leq fA \leq 1$ [f is the ratio of sequential work and total work, A is the ratio of total work and work along the longest path and sequential work ≤ length of longest path. We can show that, with processor sharing schedule and $n \geq 2$,

$$S(n) \geq \frac{nA}{n + A - 1 - (1 - fA)}$$

And, also

$$S(n) \leq \min\left\{A, \frac{n}{1 + f(n-1)}\right\}$$

Thus, we get improved lower and upper bounds [See the paper for proofs of these claims]

Addition of Processors

With processor sharing scheduling, an increase in the number of processors from n to kn ($k \geq 1$) affects speed up as follows:

$$\max\left\{1, \frac{kA}{(k-1)n+A}\right\} \leq \frac{S(kn)}{S(n)} \leq \min\left\{k, 1+(A-1)\frac{k-1}{kn-1}\right\}$$

Observe:

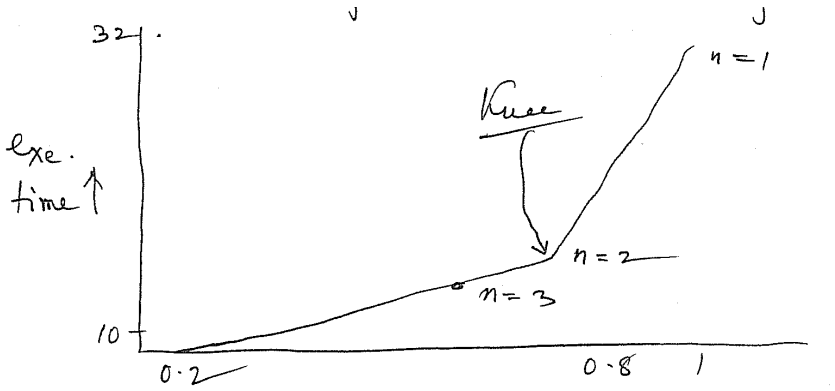
- for n = 1, these inequalities provide the speed up with k processors [using integer k, of course], – previous claims.
- if n and kn are relatively small compared to A, speed up increases by a factor close to linear in number of processors; assume n = A/9; doubling the number of processors will increase the speed up by at least 80%. If n = A, for large A, doubling n would increase speed up at most by 50%
- Uncertainty is greatest when n is close to A – if n = 2A/3 and A is large, speed up increase is anywhere between 20 and 75%

Execution Time & Efficiency Profile

This is the “cost-benefit” profile in parallel system – we need to look at the “knee” of the plot [where benefit per unit cost is maximized]. Two possible viewpoints:

- Efficiency is the benefit and execution time is the cost – system objective is to achieve efficient use of each processor while considering cost to the user in terms of increased execution time.
- Execution time is the benefit [lower the time, higher the benefit, of course] and efficiency is the cost [lower the efficiency, higher the cost]

- The profile is a graph where execution time is along y-axis and efficiency along x-axis and each point represents the combination for a given number of processors. The knee of the profile is where $E(n)/T_n$ is maximized. Here is the profile of our previous example.



- We need to know or at least estimate the knee; using a number of processors equal to the average parallelism is good estimate in the sense that it yields a point on the profile relatively close to the knee.

Multithreading programming and Parallel programming

- **Multi-core CPU** is a single CPU with two or more cores. A multi-core CPU performs multiprocessing in a single physical package (the CPU chip). A multi-core CPU provides a performance similar to a multi-CPU system but at a much lower cost since the motherboard does not need to be designed to support multiple CPUs.
- **Multitasking** is the ability of the OS to *quickly* switch between each computing task to give the impression that different applications are executing multiple actions simultaneously:
  - A system with a single one-core CPU only runs one task at a time, but multitasking allows it to give the *impression* the different applications are executing *simultaneously*.
  - A system with a single multi-core CPU can truly execute multiple tasks at the same time as each core can work on a different task. For example, a 2-core CPU can truly execute 2 tasks simultaneously, however, each core can do its own multitasking and therefore running multiple applications more efficiently to give better performance.
- **Multithreading** extends the idea of multitasking from CPUs to applications:
  - A **thread** is an execution path that runs a specific function (thread function) within a process.
  - Like multitasking, **multithreading** is the ability to the OS to to *quickly* switch between threads to give the impression that an application is executing multiple actions simultaneously.
- **Parallel** programming leverages multiple cores to improve processing. **Multithreading** is one of three required core techniques used to implement parallel programming, with the other two being **Partitioning** and **Collating**. Parallel programming is therefore programming that leverages multiple cores and all processors to improve processing speeds by:
  - *Partitioning* work into small chunks.
  - Execute these chunks in parallel using *multithreading*.
  - *Collating* the results as they become available - in a thread-safe and performant manner.

Verbatim from http://www.diranieh.com/NETCSharp/ParallelWithExamples.htm

Multithreading

- **Multithreading** extends the idea of multitasking from CPUs to applications:
  - A **thread** is an execution path that runs a specific function (thread function) within a process.
  - Similar to multitasking, **multithreading** is the ability to the OS to quickly switch between threads to give the impression that an application is executing multiple actions simultaneously.
  - Parallel programming leverages multiple cores to improve processing. Multithreading is one of three required core techniques used to implement parallel programming, with the other two being Partitioning and Collating. Parallel programming is therefore programming that leverages multiple cores and all processors to improve processing speeds by:
  - *Partitioning* work into small chunks.
  - Execute these chunks in parallel using *multithreading*.
  - *Collating* the results as they become available - in a thread-safe and performant manner.

Multicore Machines⋆

- Multicore architectures integrate multiple processing units into one chip to overcome the physical constraints of single core architectures, and their exponentially growing power consumption. Multi-core architectures provide a more cost-effective chip organization than their single core counterparts. Usually, micro-processor performance increases are roughly proportional to the square root of the increase in complexity. This means that to provide a 40% performance improvement in a single core design, we need to double the number of logical units. Multicore architectures offer a cost-effective alternative, delivering more computing capability via parallel processing, while consuming less power and board space.
- One can apply scalable computing principles that emerged in the last decade to multicore architectures and the simple hardware model proposed by Hill and Marty. Multicore architectures are fundamentally scalable and not limited by Amdahl's law. In addition to reevaluating the future of multicore scalability, one can arguably show that performance of multicore systems will ultimately be decided by the so-called memory wall.
- We assume that the multicore architecture under study is a symmetric architecture, and assume each core has its own L1 cache, where the memory bound is the cumulated capacity of the L1 caches. Should the capacity of L2 increases proportionally with the number of cores, the following analyses for L1 can be directly applied to L2.

A Simple Cost Model

- A multicore chip of given size and technology generation can contain at most n base core equivalents, where a single BCE implements the baseline core. This limit comes from the resources a chip designer is willing to devote to processor cores (with L1 caches) [not including chip resources expended on shared caches, interconnection networks, memory controllers, and others.
- We do not consider what limits a chip to **n** BCEs – like power, area, or some combination of power, area, and other factors.
- We assume that (micro-) architects are capable to use the resources of multiple BCEs to create a core with greater sequential performance. If performance of a single-BCE core is 1, assume that a more powerful core, made by using resources of r BCEs, has a sequential performance perf(r).
  - Architects should always increase core resources when perf(r) > r because doing so speeds up both sequential and parallel execution.
  - When perf(r) < r, however, the tradeoff begins. Increasing core performance aids sequential execution but hurts parallel execution.

* "Amdahl's Law in the Multicore Era", IEEE Computer, July 2008, 33 – 38.

Fixed-size speedup model of multicore

$$Speedup = \frac{\text{Enhanced performance}}{\text{Original performance}} = \frac{T_{original}}{T_{enhanced}}$$

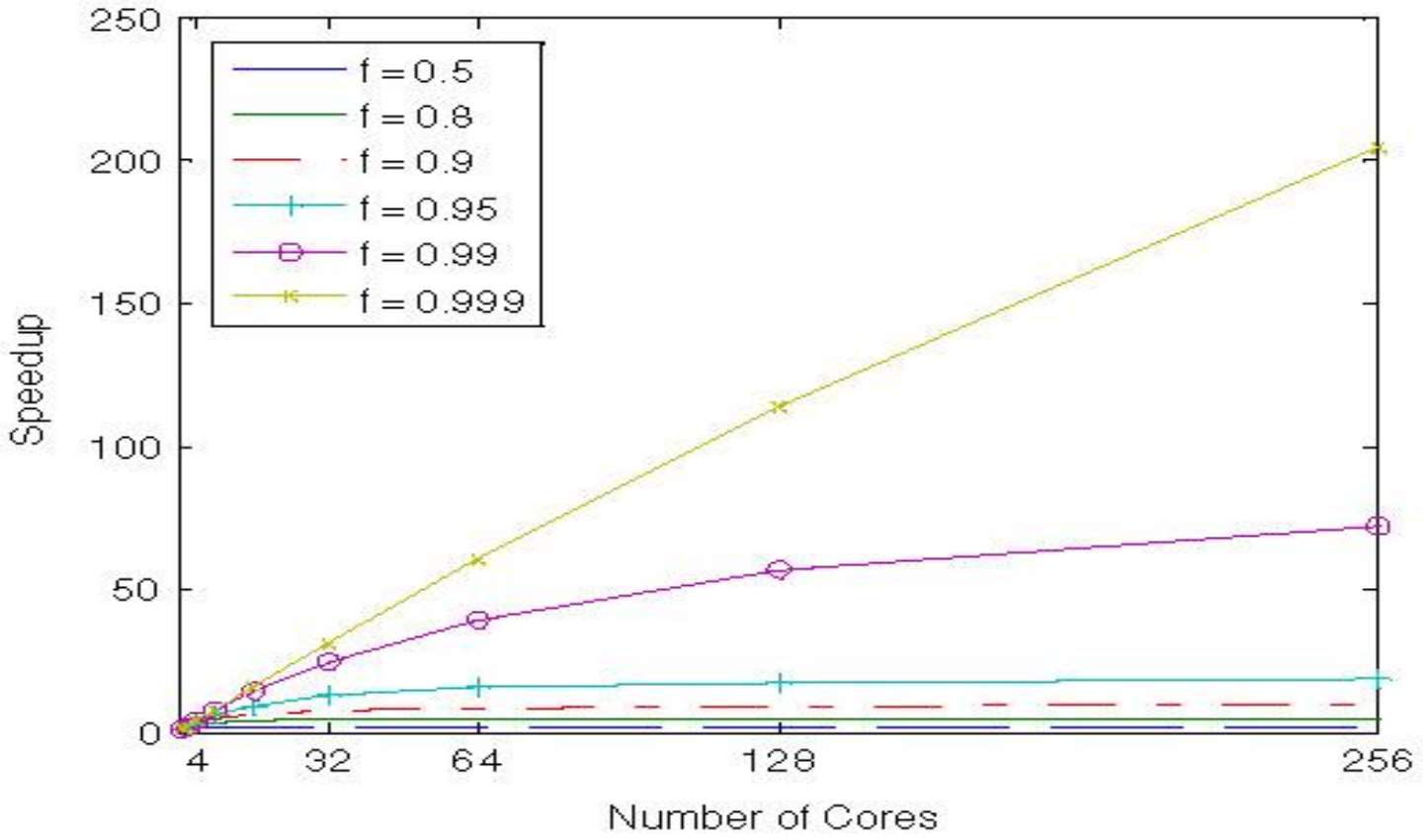
- Note that the performance is the reciprocal of the execution time. Let us assume that the problem size is w (fixed). Thus, the original execution time is $T_{Original}$ = w/perf(1) = w, where a single BCE core has a performance of 1. If we assume these n-BCE resources are built into n/r cores, where each core has a perf(r) performance, the new execution time of n-BCE multicore is

$$T_{Enhanced} = \frac{(1-f)w}{perf(r)} + \frac{f \cdot r}{(n/r) \cdot perf(r)}$$

$$Speedup = \frac{w / perf(1)}{\frac{(1-f)w}{perf(r)} + \frac{fwr}{n \cdot perf(r)}} = \frac{1}{\frac{1-f}{perf(r)} + \frac{fr}{n \cdot perf(r)}} = \frac{c}{(1-f) + \frac{f}{m}}$$

- Note that perf(r) is a constant (say *c*) for a given design and let *m = n/r* [In most designs, perf(r) ≈ √r;]. We again get the original Amdahl's law. The scalability of multicore architectures is rather limited; the speedup is quickly restricted by the sequential portion of a problem under study.

Fixed-size speedup of a multicore architecture.



The scalability of multicore architectures is rather limited – the fixed-size speedup model (Amdahl's law) illustrates a very limited scalability of a multicore architecture, and the speedup is quickly restricted by the sequential portion of a problem under study. The scalability is acceptable only when the problem is highly parallelizable, such as the improvable portion being over 99.9%.

Fixed-time speedup model

- We take n, the number of base cores, as the scaling factor. The scalability question is whether we should have a large n. Substituting n = r (base point), we have

$$\text{Speedup} = \left. \frac{1}{\frac{1-f}{perf(r)} + \frac{fr}{n \cdot perf(r)}} \right|_{n=r} = \text{perf}(r)$$

- Let n = r be the initial point, and n = mr be the scaled number of cores. Following the fixed-time model assumption that the scaling is only at the parallel portion, for the fixed-time speedup model we have:

$$\frac{(1-f)w}{perf(r)} + \frac{fw}{perf(r)} = \frac{(1-f)w}{perf(r)} + \frac{fw'}{perf(r).\text{m}} \quad \text{or } w' = mw$$

- Hence, the scaled speedup, compared with n = r is

$$speedup = \frac{Sequential\ time\ for\ w'}{Sequential\ time\ for\ w} = \frac{\frac{(1-f)w}{perf(r)} + \frac{fw'}{perf(r)}}{\frac{w}{perf(r)}} = (1-f) + mf$$

- Thus, multicore architectures are scalable under the scalable computing model, and their fixed-time speedup grows linearly with the scaling factor.

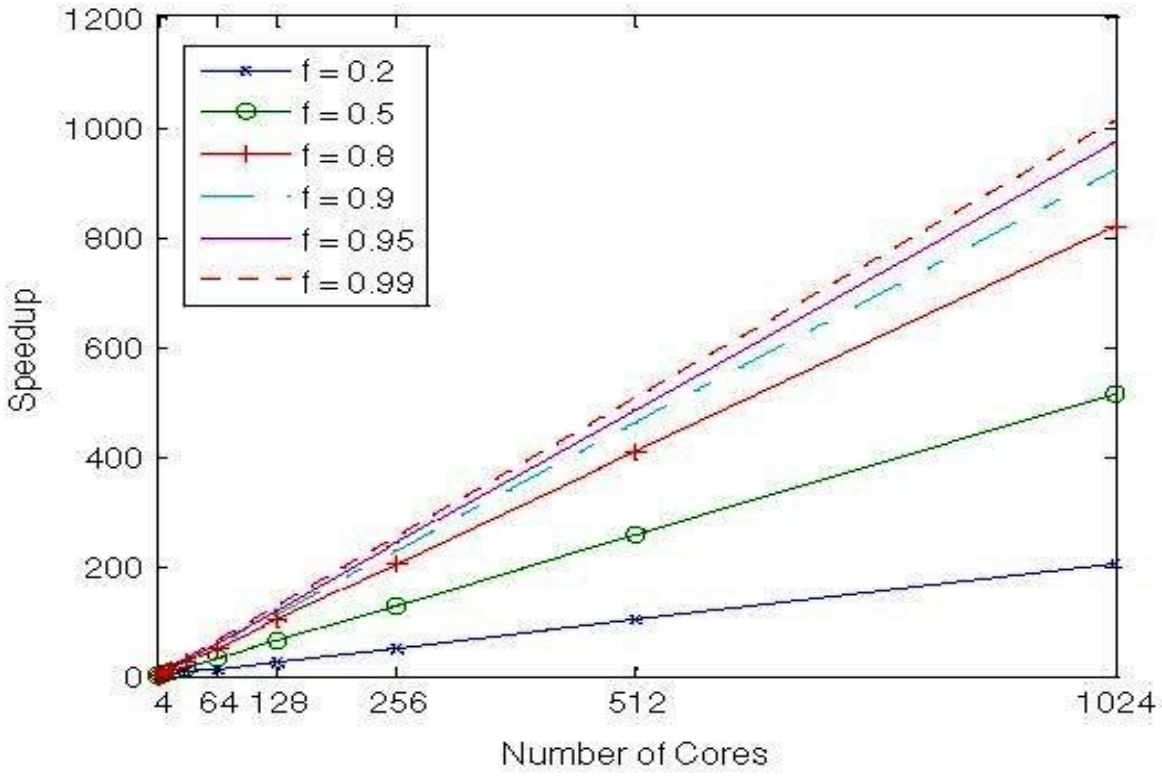
Fixed-time speedup model

- The fixed-time speedup model, as shown, presents a more optimistic view of the multicore architecture. For instance, when f equals 0.9, the speedup achieved is 922 with 1024 cores, whereby Amdahl's law, the speedup is around 10. When f = 0:99, the fixed-time speedup is 1013 with 1024 cores.
- The continued performance improvement of fixed-time speedup is due to the fact that it continuously has enough work for parallel processing. After scaling, the parallel work is fw′ , and the total work is: (1 – f)w + fw′ = [1 + (m – 1)]w. Thus, the new parallel work over total work ratio is

$$f' = \frac{mfw}{[1+(m-1)f]w} = \frac{f}{1/m+(m-1)f/m}$$

- When m → ∞, the parallel work ratio approximates to 1. Under the fixed-time model, multicore architectures are scalable and not limited by the sequential processing term.

Fixed-time speedup model



Fixed-time speedup of a multicore architecture.

Memory-bounded speedup model (multicore)

- Following a similar analysis of fixed-time model, and assuming the scaled workload under memory capacity constraint is w*, we have the speedup under memory-bounded model, when the number of cores is scaled from r to mr, as:

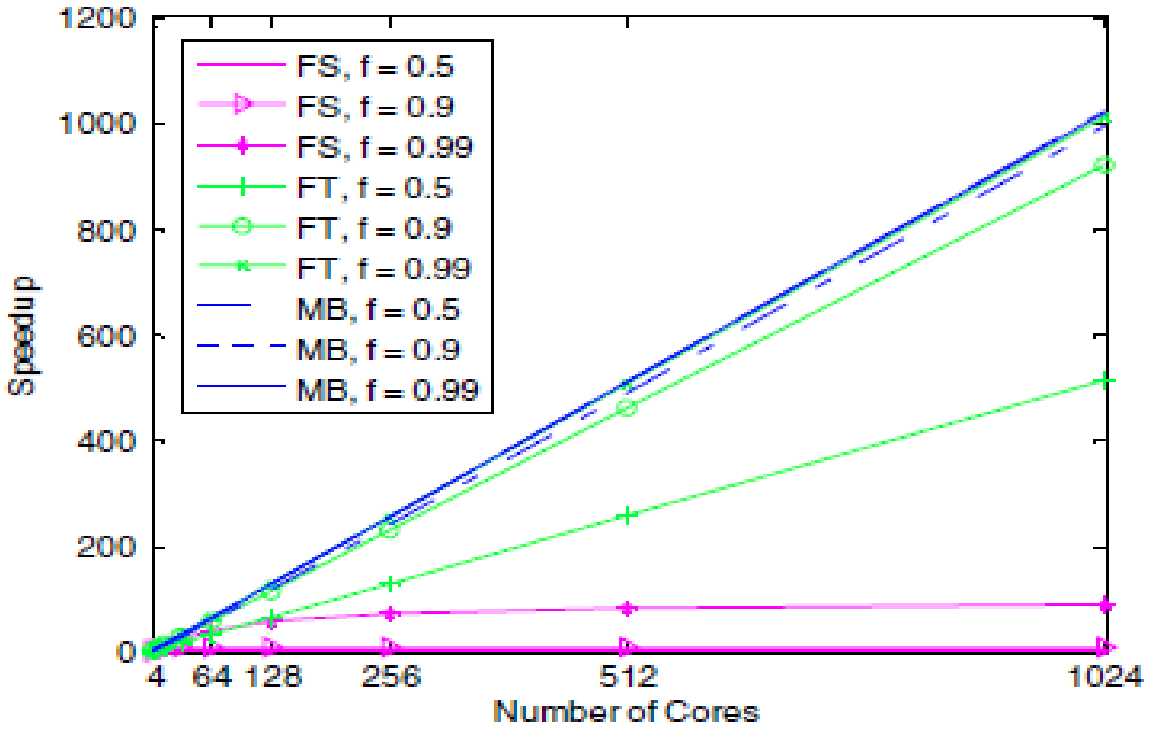
$$Speedup_{MB} = \frac{\text{Sequential Time of Solving w*}}{\text{Parallel Time of Solving w*}} = \frac{\frac{(1-f)w}{perf(r)} + \frac{fw*}{perf(r)}}{\frac{(1-f)w}{perf(r)} + \frac{fw*}{m \cdot perf(r)}} = \frac{(1-f)w + fw*}{(1-f)w + \frac{fw*}{m}}$$

- Assume y = g(x) is the function of computing requirement in terms of memory requirement, w =g(M), and assume g(x) is a power function. Therefore, following the previous analysis, the memory-bounded speedup is:

$$Speedup_{MB} = \frac{(1-f)w + f\alpha(m)w}{(1-f)w + \frac{f \cdot \alpha(m)w}{m}} = \frac{(1-f) + f \cdot \alpha(m)}{(1-f) + \frac{f \cdot \alpha(m)}{m}}$$

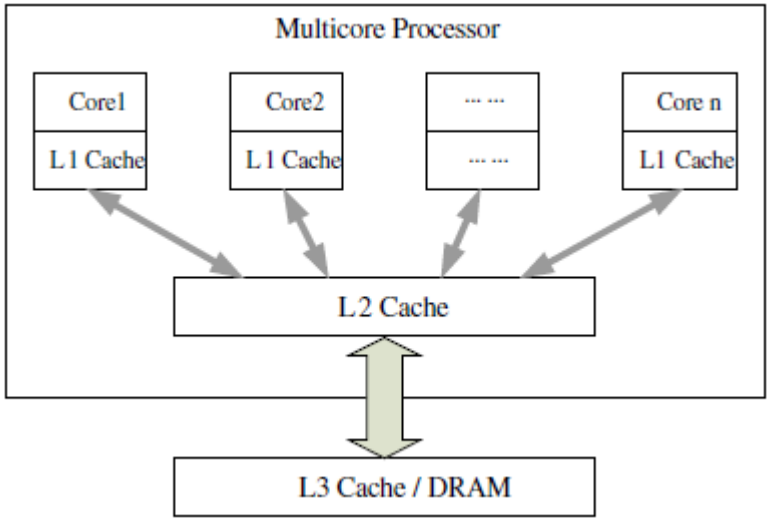
- Thus, a multicore architecture can scale up well as long as the workload size of the application can be allowed to grow with the number of cores; an even better performance can be achieved when the memory capacity constraint is used to scale the workload instead of the execution time constraint. The memory-bounded speedup model reflects situations where memory capacity is the constraint, in the case of the L1 cache of multicore architectures.

Comparisons & Lessons



- The scalable computing concept and two scaled speedup models, fixed-time speedup model and memory-bounded speedup model, are applicable to multicore architecture design.
- Sequential processing is not a limiting factor of multicore scalability, at least not in the sense of scalable computing.
- if sequential processing is not the limiting factor for scalability, then what is? Is it data access delay, or the so-called memory-wall problem. Let us revisit the scalability problem considering data access as the factor limiting performance.

Memory wall and multicore architecture scalability



- Multicore processor scalability is not necessarily the same as multicore-processor parallel processing scalability. For many applications, such as meta-tasks, high-throughput computing, or perfectly parallel applications, the sequential portion of the parallel workload is not the limiting factor for performance. Nonetheless, the performance of these types of applications is often limited on multicore architectures. Why, and what can one do?
- Consider the above architecture. The memory-bounded speedup model gives a performance upper bound where all the data are stored in the L1 caches. But, for any actual application with reasonable size, data may have to be accessed through the memory hierarchy, where long data-access delay occurs, a.k.a. the memory-wall problem*, in addition to the contention of the shared L2 cache and data paths to the lower level of the memory hierarchy.

*W.A. Wulf, S.A. McKee, Hitting the memory wall: Implications of the obvious, Computer Architecture News 23 (1) (1995) 20 – 24.

Memory Wall

- The memory-wall problem is due to the disparity of technology advance between CPU speed and memory data access latency. During last three decades memory latency in terms of processor cycles has increased roughly from 0.3 cycles in 1980 to 220 cycles in 2005, and the gap is still growing steadily.

Multicore architecture scalability

- We study the scalability of multicore architecture with a simplistic assumption: we assume data-access delay as the scalability overhead (the assumption of fixed data-access time is not true under today's technology, not yet anyway!).
- Change the cost model slightly: assume data processing work, w_p, and data communication (access) work, w_c , and $w = w_p + w_c$. We assume w_c is a function of r, but it is independent of the workload and the number of cores. As before, the design choice is to choose an appropriate r to optimize perf(r) under the same assumption that the performance of a single BCE core is 1, and the scalability concern is on determining an appropriate number of base cores, n, for best performance. Similarly, as before, fixed size speedup is

$$Speedup_{FS} = \frac{1}{\frac{w_c}{perf(r)} + \frac{w_p \cdot r}{perf(r) \cdot n}}$$

Multicore architecture scalability (contd.)

- For fixed-time speedup, taking n = r as the initial point, following the fixed-time principle where n = mr, we have:

$$\frac{w_c}{perf(r)} + \frac{w_p}{perf(r)} = \frac{w_c}{perf(r)} + \frac{w'_p}{m \cdot perf(r)} \text{ or } w'_p = m \cdot w_p$$

- Therefore, the fixed-time speedup compared with r BCEs is:

$$Speedup_{FT} = \frac{\frac{w_c}{perf(r)} + \frac{w'_p}{perf(r)}}{\frac{w_c}{perf(r)} + \frac{w_p}{perf(r)}} = \frac{w_c + m \cdot w_p}{w_c + w_p} = (1 - f') + mf' \text{ where } f' = \frac{w_p}{w_c + w_p}$$

- Similarly, $Speedup_{MB}$ can be computed as before.
- It is likely that the memory-bounded speedup is greater than the fixed-time speedup since the computing requirement is generally greater than memory requirement.
- The scalability issues of multicore architecture involve the whole architecture design of a computing system.
- With research and technology advance, we should be able to mitigate the memory-wall effect and provide a much better performance than that offered by today's multicore architecture.

References:

References:

- Xian-He Sun, Yong Chen, “Reevaluating Amdahl's law in the multicore era”, J. Parallel Distrib. Comput. 70 (2010) 183 – 188.
- Mark D. Hill, Michael R. Marty, “Amdahl’s Law in the Multicore Era”, IEEE Computer Magazine, July 2008, pp. 33 – 38.
- Dong Hyuk Woo and Hsien-Hsin S. Leepp, “Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era”, IEEE Computer Magazine, December 2008, 24 – 31.