Performance Metrics for Parallel Programs

Content

- measuring time
- towards analytic modeling,
- execution time,
- overhead,
- speedup,
- efficiency,
- cost,
- granularity,
- scalability,
- roadblocks,
- asymptotic analysis

Timing

- In order to parallelize a progr./alg., we need to know which parts of a program need the most computation time.
- Three different time spans to be considered:

1. *wall time*:
   - The time span a “clock on the wall” would measure, which is the time elapsed between start and completion of the program.
   - This is usually the time to be minimized.
2. *user time*:
   - The actual runtime used by the program.
   - this << the wall time since the program has to wait a lot, for example for computation time allocation or data from the RAM or from the hard-disk.
   - These are indications for necessary optimizations.
3. *system time*:
   - Time used not by the program itself, but by the operating system, e.g. for allocating memory or hard disk access.
   - System time should stay low.

Measuring time

- Unix command time: **time ./shale**
  - Output example:
    - real 3m13.535s
    - user 3m11.298s
    - sys 0m1.915s
  - measures the total runtime used by the program
- For the performance analysis, we want to know the **runtime required by individual parts** of a program.
  - There are several programming language and operating system dependent methods for measuring time inside a program.
    - MPI & OpenMP have their own, platform independent functions for time measurement.
      - MPI_Wtime() & omp_get_wtime() return the wall time in secs, the difference between the results of two such function calls yields the runtime elapsed between the two function calls.
- advanced method of performance analysis: **profiling.**
  - the program has to be built with information for the profiler.
  - Example: gprof
    - gprof *program* > prof.txt creates a text file with the profiling information.
      1. *flat profile* lists all function/pocedure calls, time used for them, percentage of the total time, no. of calls etc
      2. *call tree*, a listing of all procedures call by the procedures of the program.

Towards analytical modeling of parallel Progs

- A sequential algorithm is usually evaluated in terms of its execution time, expressed as a function of the size of its input.
- The execution time of a parallel algorithm depends not only on input size but also on
  1. the number of PEs used,
  2. their relative computation speed
  3. interprocess communication speed.

  $\Rightarrow$ a parallel alg. cannot be evaluated in isolation from a parallel architecture without some loss in accuracy.
- A number of measures of performance are intuitive:
  - the wall-clock time taken to solve a given problem on a given parallel platform.
  - how much faster the parallel program runs with respect to the serial program.

Execution time

- The **serial runtime** T_S of a program
  - is the time elapsed between the beginning and the end of its execution on a sequential computer.

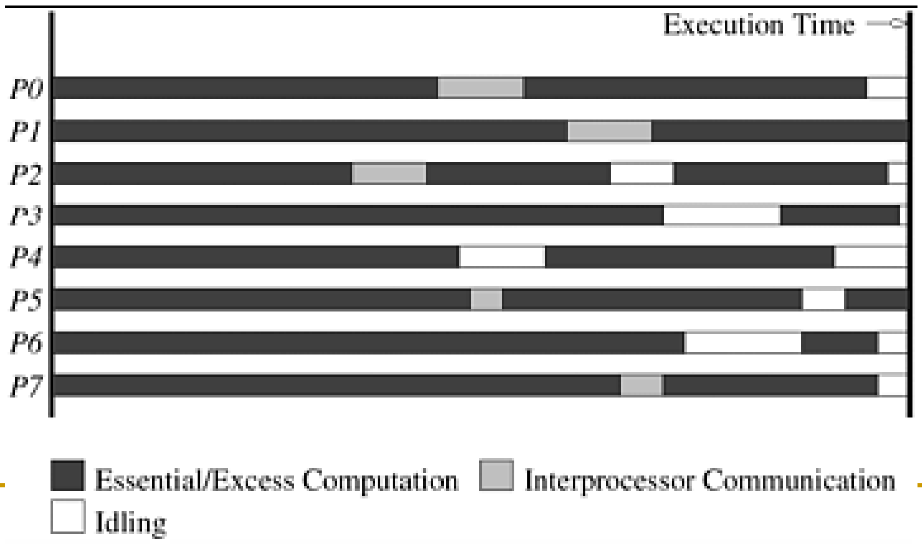
- The **parallel runtime** T_P
  - is the time that elapses from the moment a parallel computation starts to the moment the last PE finishes execution.

Factors influencing the performance

- The algorithm itself must be parallelizable & the data set to which it is to be applied must be such that an appropriately large number of processors can be applied.
- Overheads related to synchronization and memory access conflicts can lead to performance deterioration.
- Load balancing is usually rather difficult to achieve and the lack of it results in performance deterioration.
- Creation of algorithms that can be used on multiple processors often leads to the increase of computational complexity of the parallel algorithm over the sequential one.
- Dividing data among multiple memory units may reduce the memory contention and improve data locality, resulting in performance improvement.

Sources of overheads in parallel progs

- Using twice as many hardware resources, one can reasonably expect a program to run twice as fast – This is rarely the case!
- The execution profile of a hypothetical parallel program executing on 8 PEs processing elements.
  - Profile indicates times spent performing computation (both essential and excess), communication, and idling.



Sources of overheads

1. ***Interprocess interaction:***
   - Any nontrivial parallel system requires its processing elements to interact and communicate data (e.g., intermediate results).
   - The time spent communicating data between processing elements is usually the most significant source of parallel processing overhead.
2. ***Idling:***
   - due to many reasons such as:
     - load imbalance,
     - synchronization,
     - presence of serial components in a program.
   - Example1 :
     - when tasks generation is dynamic it is impossible (or at least difficult) to predict the size of the subtasks assigned to various processing elements.
     - ⇒ the problem cannot be subdivided statically among the processing elements while maintaining uniform workload.
     - ⇒ If different processing elements have different workloads, some PEs may be idle during part of the time that others are working on the problem.

Sources of overhead

2. ***Idling:***
   - Example 2: PEs must synchronize at certain points during parallel prog.execution.
     - If all processing elements are not ready for synchronization at the same time, then the ones that are ready sooner will be idle until all the rest are ready.
   - Example 3: Parts of an algorithm may be unparallelizable, allowing only a single PE to work on it.
     - While one PE works on the serial part, the other PEs must wait.

3. ***Excess Computation:***
   - Fastest sequential algorithm for a problem may be difficult/impossible to parallelize
     - use a parallel algorithm based on a poorer but easily parallelizable sequential algorithm.
       - that is, one with a higher degree of concurrency
   - Difference in computation performed by the parallel program and the best serial program = excess computation overhead incurred by the parallel program.
   - A parallel algorithm based on the best serial algorithm may still perform more aggregate computation than the serial algorithm.
   - Example: Fast Fourier Transform algorithm.
     - In its serial version, the results of certain computations can be reused.
     - Parallel version: these results cannot be reused: they are generated by different PEs.
     - Therefore, some computations are performed multiple times on different PEs.

Total parallel overhead

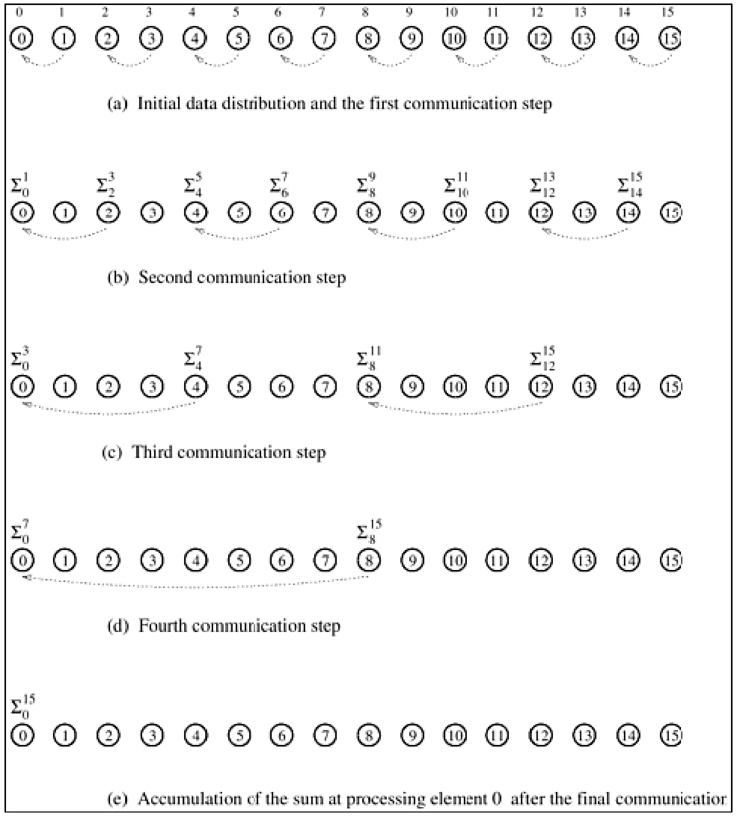
- The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function.
- We define **overhead function** or **total overhead of a parallel system,** T_o, as the total time collectively spent by all the PEs over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element.
- Consider
    - The total time spent in solving a problem summed over all processing elements is pT_P .
    - T_S units of this time are spent performing useful work,
    - The remainder is overhead
    - ⇒ the overhead function is given by $T_o = pT_P - T_S$.

Speedup

- Question: how much performance gain is achieved by parallelizing a given application over a sequential implementation?
- Speedup is a measure that captures the relative benefit of solving a problem in parallel.
- **Speedup**, S, is the ratio of the time taken to solve a problem on a single PE to the time required to solve the same problem on a parallel computer with *p* identical PEs.
  - Same type of PE in the single and parallel execution
  - speedup *S* as the ratio of the serial runtime of the *best sequential algorithm* for solving a problem to the time taken by the parallel algorithm to solve the same problem on *p* processing elements
    - Sometimes, the best sequential algorithm to solve a problem is not known,
    - Or its runtime has a large constant that makes it impractical to implement.
      - In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm.

Example: adding *n* numbers using *n* PEs

- If $n = 2^k$, perform the operation in $\log n = k$ steps
- n=16:



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(a) Initial data distribution and the first communication step
Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}
(b) Second communication step
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(c) Third communication step
Σ_0^7 Σ_8^{15}
(d) Fourth communication step
Σ_0^{15}
(e) Accumulation of the sum at processing element 0 after the final communication

$T_P = \Theta(\log n)$.
$T_S = \Theta(n)$
$S = \Theta(n/\log n)$.

Example: sorting

- Consider the example of parallelizing bubble sort.
- Assume:
  - serial version of bubble sort of 10^5 records takes 150 s
  - a serial quicksort can sort the same list in 30 s.
  - a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on 4 PEs:
- It would appear that the parallel odd-even sort algorithm results in a speedup of 150/40=3.75.
- This conclusion is misleading!
- The parallel algorithm results in a speedup of 30/40 = 0.75 with respect to the best serial algorithm.

Theoretically S<=p

- If the best sequential algorithm takes T_S units of time to solve a given problem on a single PE, then a *S* of *p* can be obtained on *p* PEs if none of the PEs spends more than time T_S/p.
- Assume: $S>p$ => possible only if each PE spends less than time T_S/p solving the problem => a single PE could emulate the *p* PEs and solve the problem in fewer than T_S units of time => This is a contradiction because *S* is computed with respect to the best sequential algorithm.

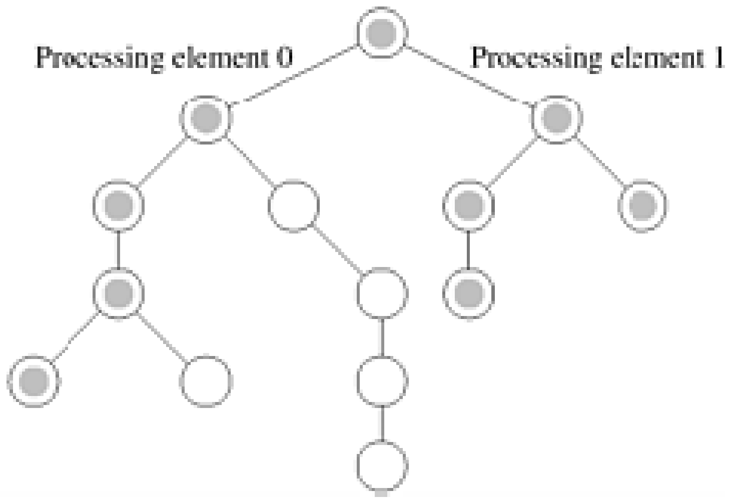
Superlinear speedup

- In practice, a speedup greater than *p* is sometimes observed.
- This usually happens:

1. When the work performed by a serial algorithm is greater than its parallel formulation
   - Exemple: search
2. Due to hardware features that put the serial implementation at a disadvantage.
   - For example:
     - the data for a problem might be too large to fit into the cache of a single PE,
     - ⇒ degrading its performance due to the use of slower memory elements.
     - ⇒ when partitioned among several PE, the individual data-partitions would be small enough to fit into their respective PE' caches.

Superlinearity due to exploratory decomposition

- Consider
  - an algorithm for exploring leaf nodes of an unstructured tree
  - each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'.
  - two processing elements using depth-first traversal.
  - Processor 0 searching the left subtree
  - Processor 1 searching the right subtree
  - Expands only the shaded nodes before the solution is found.
  - The corresponding serial formulation expands the entire tree.
- The serial algorithm does more work than the parallel algorithm.



Efficiency

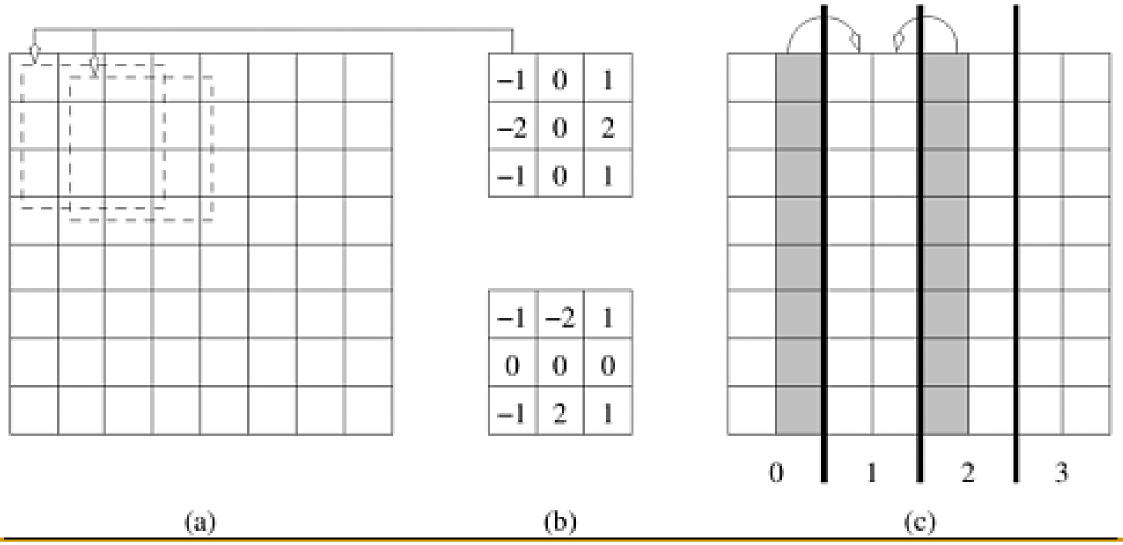
- Ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm.
  - Example: part of the time required by the PEs to compute the sum of n numbers is spent idling (and communicating in real systems).
- Efficiency is a measure of the fraction of time for which a PE is usefully employed.
- ***E* is the ratio of *S* to the no.of PEs : *E*=*S*/*p*.**
- In an ideal parallel system efficiency is equal to one.
- In practice, efficiency is between zero and one

Examples

- *Efficiency of adding n numbers on n processing elements:* E= $\Theta(n/\log n)/n=\Theta(1/\log n)$.
- Edge detection on images
  - Sequential:
    - Given an n x n pixel image, the problem of detecting edges corresponds to applying a 3x 3 template to each pixel.
      - The process of applying the template corresponds to multiplying pixel values with corresponding template values and summing across the template (a convolution operation).
      - We have nine multiply-add operations for each pixel,
      - If each multiply-add takes time t_c, then the entire operation takes time $9t_cn^2$ on a serial computer.

Edge detection - parallel

- Partitions the image equally across the PEs
- Each PE applies the template to its own subimage.
- For applying the template to the boundary pixels, a PE must get data that is assigned to the adjoining PE.
- If a PE is assigned a vertically sliced subimage of dimension n x (n/p),
  - it must access a layer of n pixels from the PE to the left & similar for the right
- The algorithm executes in two steps:
  1. exchange a layer of n pixels with each of the two adjoining processing elements - $2(t_s + t_w n)$.
  2. apply template on local subimage: $9t_c n^2/p$
- The total time for the algorithm is therefore given by: $T_P = 9t_c n^2/p + 2(t_s + t_w n)$.
- $S = 9t_c n^2 / [9t_c n^2/p + 2(t_s + t_w n)]$, $E = 1/[1 + 2(t_s + t_w n)p / 9t_c n^2]$.



Cost

- = parallel runtime x the no. of PEs used
- Cost reflects the sum of the time that each PE spends solving the problem
  - ⇒ *E* can also be expressed as the ratio of the execution time of the fastest known sequential alg. for solving a problem to the cost of solving the same problem on *p* PEs.
- p=1: The cost of solving a problem on a single PE is the execution time of the fastest known sequential algorithm.
- A parallel alg. is said to be ***cost-optimal*** if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Θ terms) as a function of the input size as the fastest-known sequential algorithm on a single PE.
  - ⇒ Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel alg. has an efficiency of Θ(1).

Examples

- ***Cost of adding n numbers on n processing elements.***
  - Cost= $\Theta(n \log n)$.
  - The serial runtime of this operation is $\Theta(n)$ => the algorithm is not cost optimal.
- ***Sorting alg.***
  - Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$.
  - Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n/\log n$ and $1/\log n$, respectively.
  - Cost= $n(\log n)^2$ => this algorithm is not cost optimal

Effect of Granularity on Performance- Theory

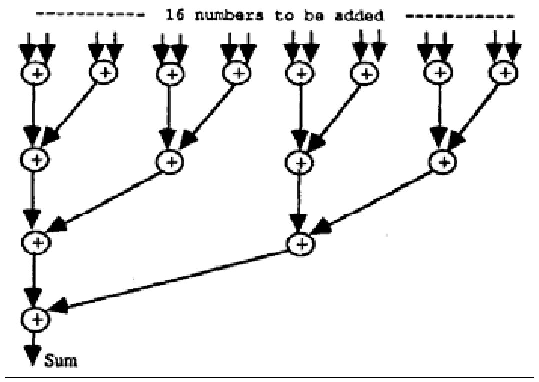
- Adding n no. with n PEs - excessive in terms of the number of processing elements.
- In practice, we assign larger pieces of input data to PEs.
  - This corresponds to increasing the granularity of computation on PEs.
- Using fewer than the maximum possible no. of PEs to execute a parallel algorithm is called **scaling down** a parallel system in terms of the no.of PEs.
- A **naive way** to scale down a parallel sys. is to design a parallel algorithm for one input element per PE, and then use fewer PEs to simulate a large no. of PEs.
  - If there are *n* inputs and only *p* processing elements ($p < n$), we can use the parallel alg. designed for *n* PEs by assuming *n* virtual PEs and having each of the *p* physical PEs simulate *n/p* virtual PEs.
  - The total parallel runtime increases, at most, by a factor of *n/p*, and the processor-time product does not increase.

    => If a parallel system with *n* PEs is cost-optimal, using *p* PEs (where $p < n$) to simulate *n* PEs preserves cost-optimality.

Effect of Granularity on Performance -practice

- Adding example: Consider $p << n$.
  - Assign n tasks to $p < n$ PEs => a parallel time less than $n(\log n)^2/p$.
  - The corresponding speedup of this formulation is $p/\log n$.
  - Examples:
    - sorting 1024 numbers (n = 1024, log n = 10) on p=32 PEs=> S=3.2.
    - $n = 10^6$, log n = 20 => S=1.6. Worst!

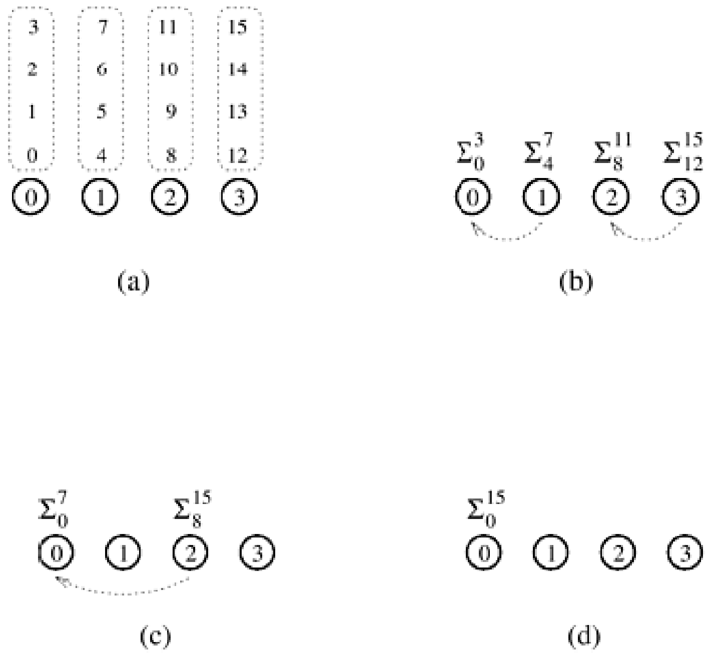
Remark: if a parallel system is not cost-optimal to begin with, it may still not be cost-optimal after the granularity of computation increases



n=16, p=8

Adding *n* numbers cost-optimally

- Example: n = 16 and p = 4.
- In the first step of this alg., each PE locally adds its n/p numbers in time $\Theta(n/p)$.
- Now the problem is reduced to adding the p partial sums on p processing elements, which can be done in time $\Theta(\log p)$ by the method described in the first example.
- The parallel runtime of this algorithm is $TP=\Theta(n/p+\log p)$
- Its cost is $\Theta(n + p \log p)$.
- As long as $n = \Omega(p \log p)$, the cost is $\Theta(n)$, which is the same as the serial runtime.
- Hence, this parallel system is cost-optimal.
- Demonstrate that the manner in which the computation is mapped onto PEs may determine whether a parallel system is cost-optimal.
- Note: We cannot make all non-cost-optimal systems cost-optimal by scaling down the no. of PEs.



3 2 1 0
7 6 5 4
11 10 9 8
15 14 13 12
0 1 2 3
(a)
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(b)
Σ_0^7 Σ_8^{15}
(c)
Σ_0^{15}
(d)

Scaling Characteristics of Parallel Programs

- $E = 1/(1+T_O/T_S)$,
  - T_o grows at least linearly with p.
  - ⇒ the overall efficiency of the parallel program goes down for a given problem size (constant T_S)

Example:

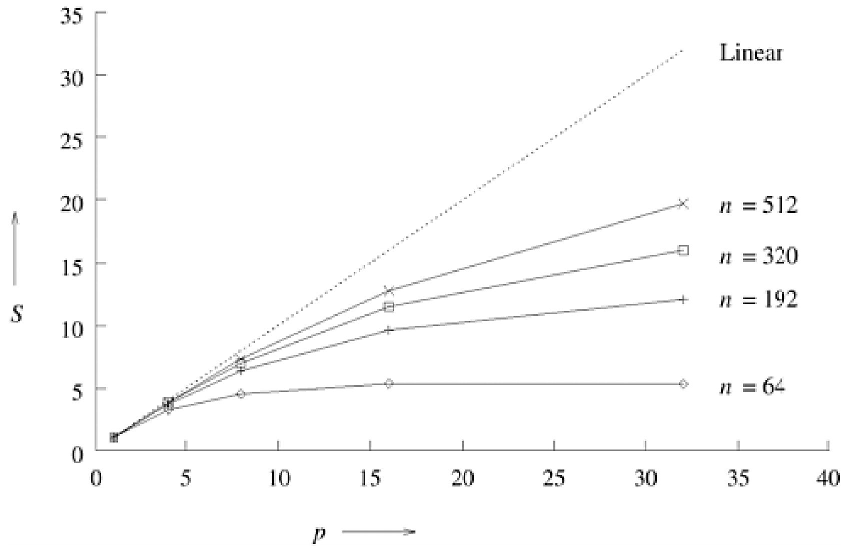
adding n no. on p PEs,

$T_P = n/p + 2 \log p$,

$S = n/(n/p + 2 \log p)$,

$E = 1/(1 + 2p \log p / n)$.

Compute S and E as functions of (n,p)



Scalable parallel systems

- The scalability of a parallel system is a measure of its capacity to increase *S* in proportion to the no. of PEs.
- Reflects a parallel system's ability to utilize increasing processing resources effectively.

- Example: + *n* no.on *p* PEs

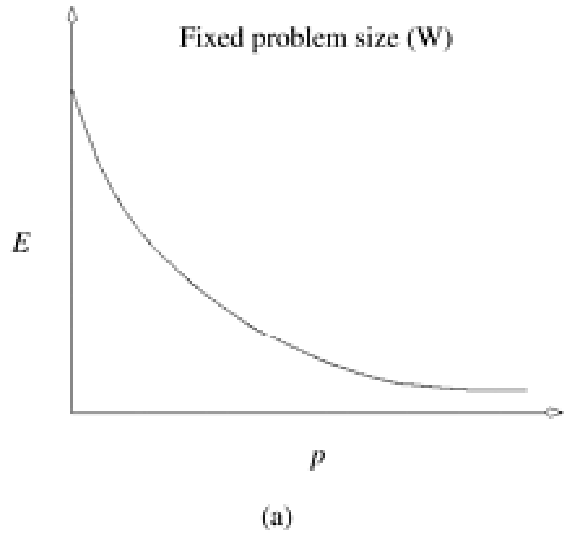
cost-optimal - $n = \Omega(p \log p)$.

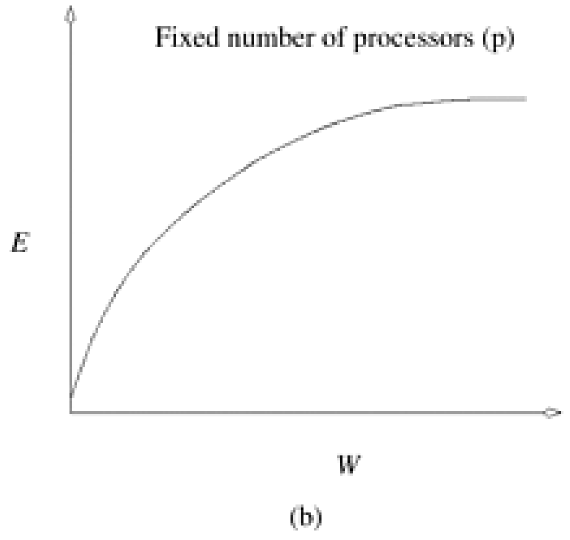
  - E=0.80 for $n = 64$ and $p = 4$, $n = 8 p \log p$.
  - $p = 8$, $n= 8 p \log p = 192$, E=0.80
  - $p = 16$, $n = 8 p \log p = 512$, E=0.80 .

=> This parallel system remains cost-optimal at an efficiency of 0.80 if *n* is increased as $8 p \log p$.

Remarks

- For a given problem size, as we increase the no. PEs, the overall efficiency of the parallel syst goes down.
- In many cases, the efficiency of a parallel syst increases if the problem size is increased while keeping the no. of PEs constant.
- a scalable parallel system= as one in which the efficiency can be kept constant as the no. of PEs is increased, provided that the problem size is also increased.





Isoefficiency Function

- Problem size: *W*.
- we assume that it takes unit time to perform one basic computation step of an alg

⇒ $W = T_S$ (of the fastest known algorithm).

⇒ $T_P=[W+T_0(W,p)]/p$, $S= W/ T_P = Wp/[W+T_0 (W,p)]$, $E=S/p= W/[W+T_0(W,p)]= 1/ [1+T_0 (W,p)/W]$.

⇒ *W* must be increased with respect to *p* to maintain *E* fixed

- Parallel syst is highly scalable if *W* need to grow linearly with respect to *p*
- Parallel syst is poorly scalable if *W* need to grow exponentially with *p*
- $K=E/(1-E)$, $W=KT_0(W,p)$ => Extract *W* as a function of *p*

This function is called isoefficiecy function

Isoefficiency function of adding numbers

- The overhead function for the problem of adding *n* numbers on *p* processing elements is approx 2*p* log *p*.
- Substituting *To* by 2*p* log *p* we get *W*=*K* 2*p* log *p*.
- Isoefficiency function for this parallel sys. is Θ(*p* log *p*).
  - If the no. of PEs is increased from *p* to *p*', the problem size *n* must be increased by a factor of (*p*' log *p*')/(*p* log *p*) to get the same *E* as on *p* PEs.
- Remark:
  - the overhead due to communication is a function of *p* only.
  - In general, communication overhead can depend on both the problem size and the no. of PEs.

Minimum execution time for adding n no.

- The parallel run time for the problem of adding n numbers on p PEs is $T_P = n/p + 2 \log p$.
- d T_P/d$p=0 \Rightarrow p=n/2$ and we get $T_P^{min} = 2 \log p$.
- Minimum cost-optimal execution time for adding n numbers:
  - minimum time in which a problem can be solved by a cost-optimal parallel system.
  - after some computations (see textbook): $T_P^{cost\_opt} = 2 \log n - \log \log n$.

Other Scalability Metrics

- Suited to different system requirements.
  - For example, in real time applications, the objective is to scale up a system to accomplish a task in a specified time bound:
    - multimedia decompression, where MPEG streams must be decompressed at the rate of 25 frames/second.
  - In many applications, the maximum size of a problem is constrained not by time, efficiency, or underlying models, but by the memory available on the machine.
    - metrics make assumptions on the growth function of available memory (with no.of PEs) and estimate how the performance of the parallel sys.changes with such scaling.

Scaled Speedup

- This metric is defined as the speedup obtained when the problem size is increased linearly with the no. of PEs .
- If the scaled-speedup curve is close to linear with respect to the no of PEs, then the parallel system is considered scalable.
- Method 1:
  - the size of the problem is increased to fill the available memory on the parallel computer.
  - The assumption here is that aggregate memory of the system increases with the no. of PEs.
- Method 2:
  - the size of the problem grows with *p* subject to an upper-bound on execution time.

Memory & time-constrained scaling

- Multiplying a matrix dimension n x n with a vector:
  - $T_S = t_c n^2$, where t_c is the time for a single multiply-add operation.
  - $T_P = t_c n^2/p + t_s \log p + t_w n$, $S = t_c n^2 / (t_c n^2/p + t_s \log p + t_w n)$.
  - Total memory requirement of the algorithm is $\Theta(n^2)$.
  - Let us consider the two cases of problem scaling.
    - memory constrained scaling:
      - we assume that the memory of the parallel system grows linearly with the no. of PEs, i.e., $m = \Theta(p) \Rightarrow n^2 = c \times p$, for some constant c.
      - The scaled speedup S' is given by: $S' = t_c c \times p / (t_c c \times p / p + t_s \log p + t_w \text{sqrt}(c \times p))$ or $S' = c_1 p / (c_2 + c_3 \log p + c_4 \text{sqrt}(p))$.
      - In the limiting case, $S' = O(\text{sqrt}(p))$.
    - time constrained scaling, we have $T_P = O(n^2/p)$. Since this is constrained to be constant, $n^2 = O(p) \Rightarrow$ this case is identical to the memory constrained case.
- Multiplying two matrices – see textbook
  - Memory constrained scaled: $S' = O(p)$.
  - Time constrained scaling: $S'' = O(p^{5/6})$

Serial fraction

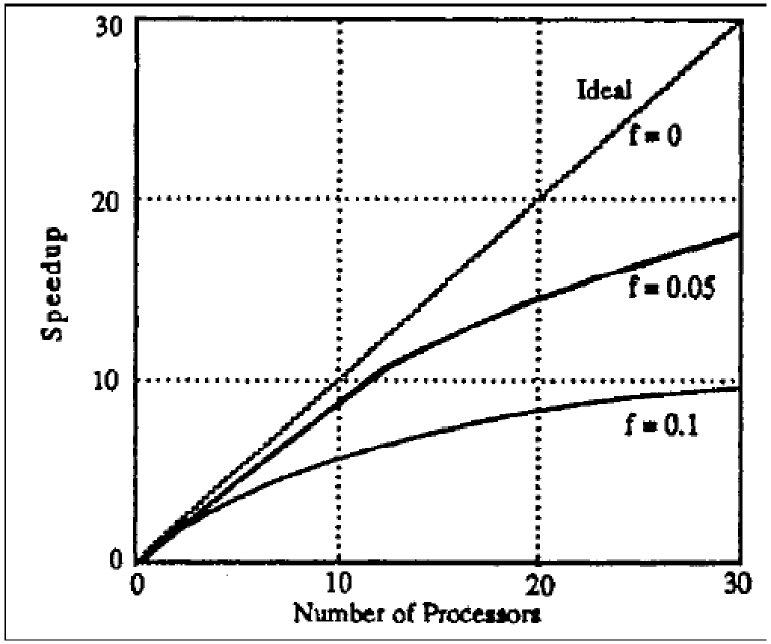
- The experimentally determined serial fraction *f* can be used to quantify the performance of a parallel system on a fixed-size problem.
- Consider a case when the serial runtime of a computation can be divided into a totally parallel and a totally serial component, i.e., $W = Tser + Tpar$
- Ideal: $T_P = Tser + Tpar/p$.
- All of the other parallel overheads such as excess computation and communication are captured in the serial component *Tser*.
- The serial fraction *f* of a parallel program is defined as: $f = Tser/W$.
- $T_P = Tser + (W - Tser.)/p \Rightarrow T_P/W = f + (1-f)/p;$
- $S = W/T_P \Rightarrow 1/S = f + (1-f)/p \Rightarrow f = (1/S - 1/p)/(1 - 1/p)$.
- Smaller values of *f* are better since they result in higher efficiencies.
- If *f* increases with the no. PEs, then it is considered as an indicator of rising communication overhead, and thus an indicator of poor scalability.
- ***Example:*** serial component of the matrix-vector product: $f = (t_s\, p \log p + t_w n\, p) / [t_c n^2 (p-1)]$ - denominator of this equation is the serial runtime of the alg. and the numerator corresponds to the overhead in parallel execution.

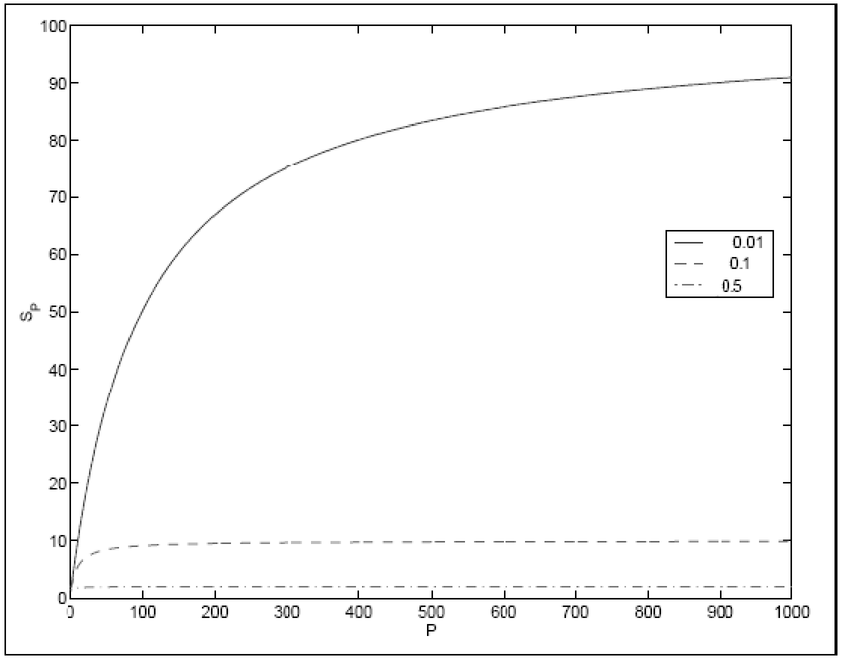
Roadblocks to Parallel Processing

Amdahl's law (1967)

- He established how slower parts of an algorithm influence its overall performance
  - since the sequential parts of an algorithm are the "slowest," Amdahl's law dictates that these parts have the most serious negative impact on the overall performance.
- States that the fraction f of inherently sequential or unparallelizable computation severely limits the speed-up that can be achieved with p processors.
- Assume: a fraction 1-f of the algorithm can be divided into p parts and ideally parallelized, the remaining f of operations cannot be parallelized and thus have to be executed on a single processor. The total
- $S = p/(fp + (1 - f))$ (see previous slide).
- Since $f < 1 \Rightarrow Sp < 1/f$.
- *the speedup achievable through parallel computing is bound by the value that is inversely proportional to the fraction of the code that has to be executed sequentially.*

Effects of Amdahl's law





If *f* = 0.1, so that 90% of an algorithm can be ideally parallelized,
and if *p* = 10, S< 6.
If *f* = 0.01, meaning only 1% of the program is not parallelizable, for
p = 100 we have that *S* = 50, so we operate at half the maximum efficiency.

Comments

- the derivation of Amdahl's law relies on the assumption that the serial work *f* is independent of the size of the problem size *n*.
  - In practice, it has been observed that *f* decreases as a function of problem size.
  - Therefore, the upper bound on the speed-up factor *S* usually increases as a function of problem size.
- Another anomaly is the so-called *superlinear speed-up*, which means that the speed-up factor has been measured to be more than *P*.
  - This may happen because of memory access and cache mismanagement or because the serial implementation on a single processor is suboptimal.
- There exist appls for which the sequential overhead is very small.
- If the original serial computation is limited by resources other than the availability of CPU cycles, the actual performance could be much better
  - A large parallel machine may allow bigger problems to be held in memory, thus reducing virtual memory paging,
  - Multiple processors each with its own cache may allow much more of the problem to remain in the cache.
- Amdahl's law assumes that for any given input, the parallel and serial implementations perform exactly the same no. of computational steps.
  - If the serial algorithm being used in the formula is not the best possible algorithm for the problem, then a clever parallel algorithm that structures the computation differently can reduce the total number of computational steps.

Gustafson’s law

- Rather than asking how fast a given serial program would run on a parallel machine, he asks how long a given parallel program would have taken to run on a serial processor.
- $T_{total}(1) = T_{setup} + pT_{compute}(p) + T_{finalization}$.
- Scaled serial fraction: $\gamma_{\mathrm{scaled}} = (T_{setup} + T_{finalization}) / T_{total}(p)$
- Then $T_{total}(1) = \gamma_{\mathrm{scaled}}\, T_{total}(p) + p(1 - \gamma_{\mathrm{scaled}})\, T_{total}(p)$.
- Rewriting the equation for speedup and simplifying:

  scaled (or fixed time) speedup: $S(P) = P + (1 - P)\, \gamma_{\mathrm{scaled}}$.
- This equation is known as Gustafson's law
- Since γ_{scaled} depends on *p*, the result of taking the limit isn't obvious, but would give the same result as the limit in Amdahl's law.
- We take the limit in *p* while holding $T_{compute}$ and thus γ_{scaled} constant.
  - The interpretation is that we are increasing the size of the problem so that the total running time remains constant when more processors are added.
  - This contains the implicit assumption that the execution time of the serial terms does not change as the problem size grows.

=> In this case, the speedup is linear in P *!* => if the problem grows as more processors are added, Amdahl's law will be pessimistic!

Other laws

1. ***Grosch's law:*** economy of scale applies, or computing power is proportional to the square of cost
   - If this law did in fact hold, investing money in p processors would be foolish as a single computer with the same total cost could offer p^2 times the performance of one such processor.
   - Grosch's law was formulated in the days of mainframes and did hold for those machines.

2. ***Minsky's conjecture:*** S is proportional to the logarithm of p
   - Roots in an analysis of data access conflicts assuming random distribution of addresses.
   - These conflicts will slow everything down to the point that quadrupling the number of processors only doubles the performance.
   - However, data access patterns in real applications are far from random.
   - Real speed-up can range from log p to p (p/log p being a reasonable middle ground).

3. ***The software inertia:*** billions of $ worth of existing software makes it hard to switch to parallel systs; the cost of converting the "decks" to parallel programs and retraining the programmers is prohibitive.
   - not all programs needed in the future have already been written.
   - new appls will be developed & new probls will become solvable with increased performance.
   - Students are being trained to think parallel.
   - Tools are being developed to transform sequential code into parallel code automatically.

Asymptotic Analysis of Parallel Programs

Evaluating a set of parallel programs for solving a given problem

- Example: sorting
- The fastest serial programs for this problem run in time O (*n* log *n*).
- Let us look at four different parallel algorithms A1, A2, A3, and A4, for sorting a given list.
- Objective of this exercise is to determine which of these four algorithms is the best.

	A1	A2	A3	A4
p	n^2	log n	n	Ön
T_P	1	n	Ön	Ön log n
S	n log n	log n	Ön log n	Ön
E	Log n /n	1	Log n/ Ön	1
pT_P	n^2	n log n	$n^{1.5}$	n log n

Sorting example

- The simplest metric is one of speed
  - the algorithm with the lowest T_P is the best.
  - by this metric, algorithm A1 is the best, followed by A3, A4, and A2.
- Resource utilization is an important aspect of practical program design
  - We will rarely have n^2 PEs as are required by algorithm A1.
  - This metric of evaluating the algorithm presents a starkly different image: algs A2 and A4 are the best, followed by A3 and A1.
- Cost:
  - Last row of Table presents the cost of the four algorithms.
  - The costs of algorithms A1 and A3 are higher than the serial runtime of $n \log n$ and therefore neither of these algorithms is cost optimal.
  - Algorithms A2 and A4 are cost optimal.
- Conclusions:
  - Important to first understand the objectives of parallel algorithm analysis and to use appropriate metrics, because use of different metrics may often result in contradictory outcomes