Analytical Modeling of Parallel Systems

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text ``Introduction to Parallel Computing'',
Addison Wesley, 2003.

Topic Overview

- Sources of Overhead in Parallel Programs
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Scalability of Parallel Systems
- Minimum Execution Time and Minimum Cost-Optimal Execution Time
- Asymptotic Analysis of Parallel Programs
- Other Scalability Metrics

Analytical Modeling - Basics

- A sequential algorithm is evaluated by its runtime (in general, asymptotic runtime as a function of input size).

- The asymptotic runtime of a sequential program is identical on any serial platform.

- The parallel runtime of a program depends on the input size, the number of processors, and the communication parameters of the machine.

Analytical Modeling - Basics

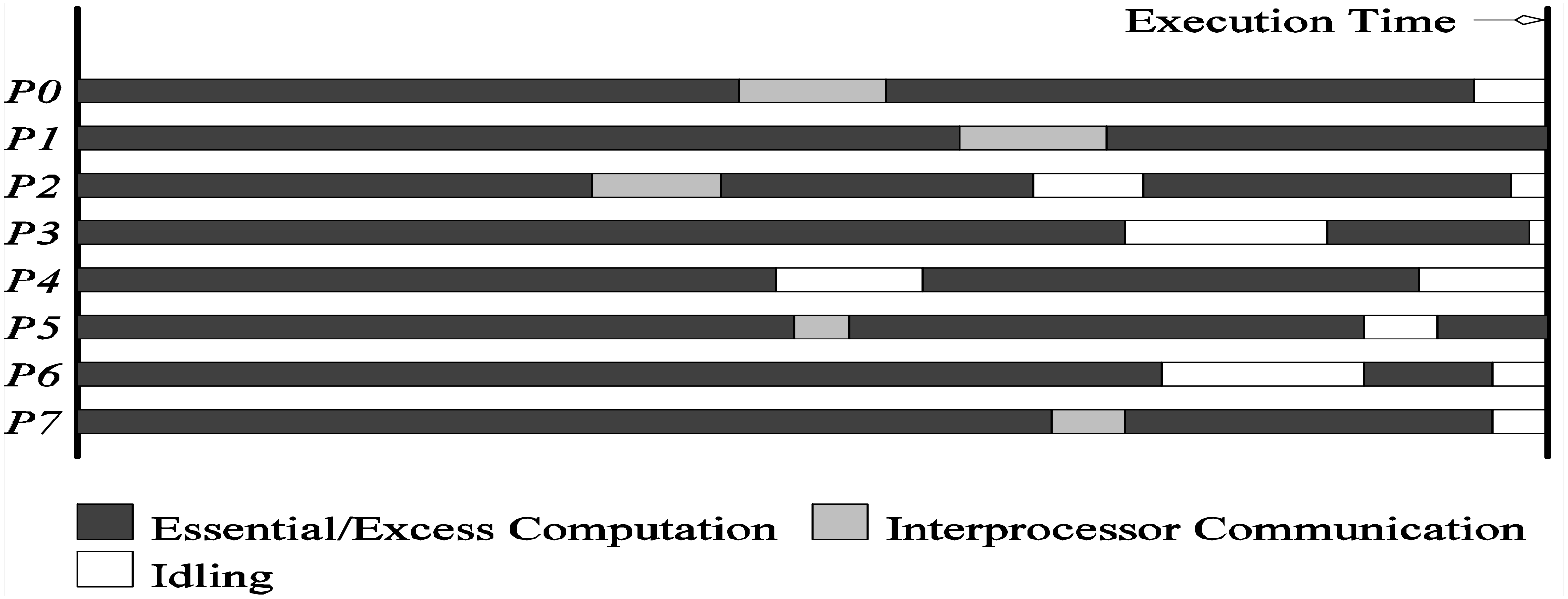
- An algorithm must therefore be analyzed in the context of the underlying platform.

- A parallel system is a combination of a parallel algorithm and an underlying platform.

- **Wall clock time** - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble.

Sources of Overhead in Parallel Programs

- If I use two processors, shouldnt my program run twice as fast?
- No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Sources of Overheads in Parallel Programs

- **Interprocess interactions:** Processors working on any non-trivial parallel problem will need to talk to each other.

- **Idling:** Processes may idle because of load imbalance, synchronization, or serial components.

- **Excess Computation:** This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Performance Metrics for Parallel Systems: **Execution Time**

- Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.

- The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

- We denote the serial runtime by T_S and the parallel runtime by T_P.

Performance Metrics for Parallel Systems: **Total Parallel Overhead**

- Let T_{all} be the total time collectively spent by all the processing elements.

- T_S is the serial time.

- Observe that $T_{all} - T_S$ is then the total time spend by all processors combined in non-useful work. This is called the *total overhead*.

- The total time collectively spent by all the processing elements $T_{all} = p\, T_P$ (***p*** is the number of processors).

- The overhead function (T_o) is therefore given by

$$T_o = p\, T_P - T_S \qquad (1)$$

Performance Metrics for Parallel Systems: **Speedup**

- What is the benefit from parallelism?
- Speedup (***S***) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

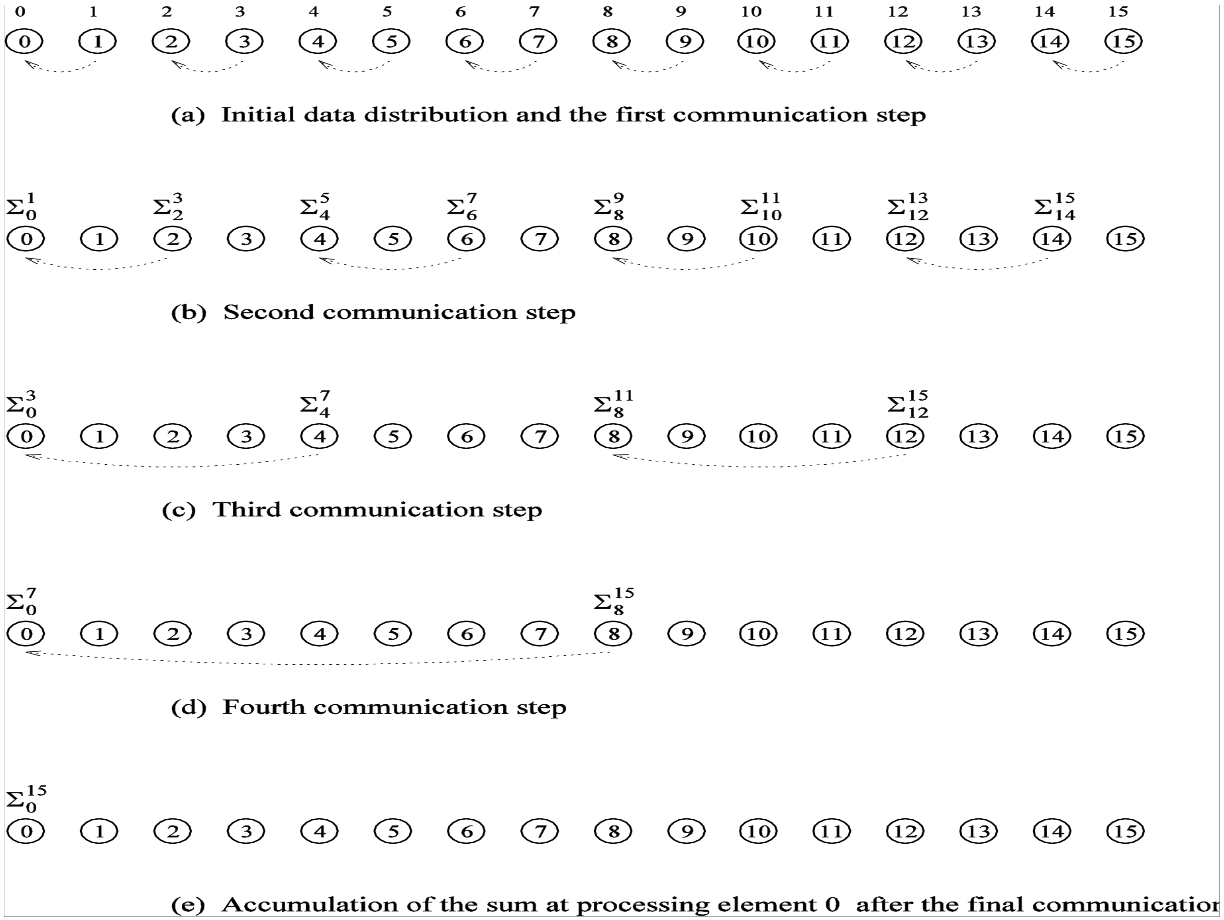
$$S = T_s / T_p$$

Performance Metrics: Example

- Consider the problem of adding ***n*** numbers by using ***n*** processing elements.

- If ***n*** is a power of two, we can perform this operation in **log *n*** steps by propagating partial sums up a logical binary tree of processors.

Performance Metrics: Example


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Σ^1_0 Σ^3_2 Σ^5_4 Σ^7_6 Σ^9_8 Σ^{11}_{10} Σ^{13}_{12} Σ^{15}_{14}
Σ^3_0 Σ^7_4 Σ^{11}_8 Σ^{15}_{12}
Σ^7_0 Σ^{15}_8
Σ^{15}_0

(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

(e) Accumulation of the sum at processing element 0 after the final communicatior

Computing the globalsum of 16 partial sums using 16 processing elements . Σ^j_i denotes the sum of numbers with consecutive labels from ***i*** to ***j***.

Performance Metrics: Example (continued)

- If an addition takes constant time, say, $\boldsymbol{t_c}$ and communication of a single word takes time $\boldsymbol{t_s} + \boldsymbol{t_w}$, we have the parallel time $\boldsymbol{T_P} = \Theta\ (\mathbf{log}\ \boldsymbol{n})$

- We know that $\boldsymbol{T_S} = \Theta\ (\boldsymbol{n})$

- Speedup $\boldsymbol{S}$ is given by $\boldsymbol{S} = \Theta\ (\boldsymbol{n} / \mathbf{log}\ \boldsymbol{n})$

Performance Metrics: Speedup

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

- For the purpose of computing speedup, we always consider the best sequential program as the baseline.

Performance Metrics: Speedup Example

- Consider the problem of parallel bubble sort.
- The serial time for bubble sort is 150 seconds.
- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds.
- The speedup would appear to be 150/40 = 3.75.
- But is this really a fair assessment of the system?
- What if serial quick sort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system.

Performance Metrics: Speedup Bounds

- Speedup can be as low as 0 (the parallel program never terminates).

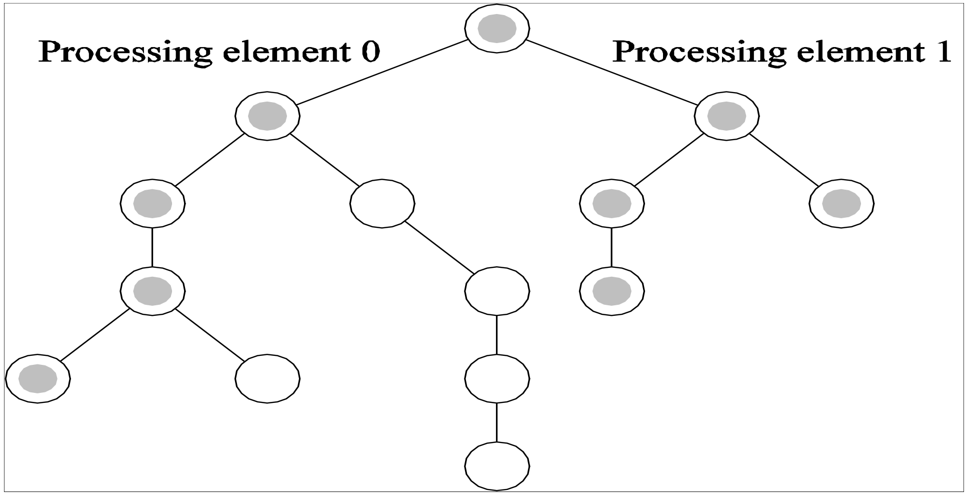
- Speedup, in theory, should be upper bounded by ***p*** - after all, we can only expect a ***p***-fold speedup if we use times as many resources.

- A speedup greater than ***p*** is possible only if each processing element spends less than time T_S / p solving the problem.

- In this case, a single processor could be time slided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Superlinear Speedups

One reason for superlinearity is that the parallel version does less work than corresponding serial algorithm.



Searching an unstructured tree for a node with a given label, `S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.

Performance Metrics:
Superlinear Speedups

Resource-based superlinearity: The higher aggregate cache/memory bandwidth can result in better cache-hit ratio, and therefore superlinearity.

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed

- Mathematically, it is given by

$$E = \frac{S}{p}. \tag{2}$$

- Following the bounds on speedup, **efficiency can be as low as 0 and as high as 1**.

Performance Metrics: Efficiency Example

- The speedup of adding numbers on processors is given by

$$S = \frac{n}{\log n}$$

- Efficiency is given by

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n}$$

$$= \Theta\left(\frac{1}{\log n}\right)$$

Cost of a Parallel System

- Cost is the product of parallel runtime and the number of processing elements used ($\boldsymbol{p}$ x $\boldsymbol{T_P}$).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $\boldsymbol{E} = \boldsymbol{T_S} / \boldsymbol{p}\ \boldsymbol{T_P}$, for cost optimal systems, $\boldsymbol{E} = O(1)$.
- Cost is sometimes referred to as ***work* or *processor-time product***.

Cost of a Parallel System: Example

Consider the problem of adding numbers on processors.

- We have, $\boldsymbol{T_P}$ = **log** $\boldsymbol{n}$ (for $\boldsymbol{p}$ = $\boldsymbol{n}$).
- The cost of this system is given by $\boldsymbol{p\ T_P}$ = $\boldsymbol{n}$ **log** $\boldsymbol{n}$.
- Since the serial runtime of this operation is Θ($\boldsymbol{n}$), the algorithm is not cost optimal.

Impact of Non-Cost Optimality

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.
- The $p\,T_P$ product of this algorithm is $n (\log n)^2$.
- This algorithm is not cost optimal but only by a factor of $\log n$.
- If $p < n$, assigning n tasks to p processors gives $T_P = n (\log n)^2 / p$.
- The corresponding speedup of this formulation is $p / \log n$.
- This speedup goes down as the problem size n is increased for a given p !

Effect of Granularity on Performance

- Often, using fewer processors improves performance of parallel systems.

- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down a* parallel system.

- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.

- Since the number of processing elements decreases by a factor of ***n*** / ***p***, the computation at each processing element increases by a factor of ***n*** / ***p***.

- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might talk to each other. This is the basic reason for the improvement from building granularity.

Building Granularity: Example

- Consider the problem of adding ***n*** numbers on ***p*** processing elements such that ***p*** < ***n*** and both ***n*** and ***p*** are powers of 2.

- Use the parallel algorithm for ***n*** processors, except, in this case, we think of them as virtual processors.

- Each of the ***p*** processors is now assigned ***n*** / ***p*** virtual processors.

- The first **log *p*** of the **log *n*** steps of the original algorithm are simulated in (***n*** / ***p***) **log *p*** steps on ***p*** processing elements.

- Subsequent **log *n*** - **log *p*** steps do not require any communication.

Building Granularity: Example (continued)

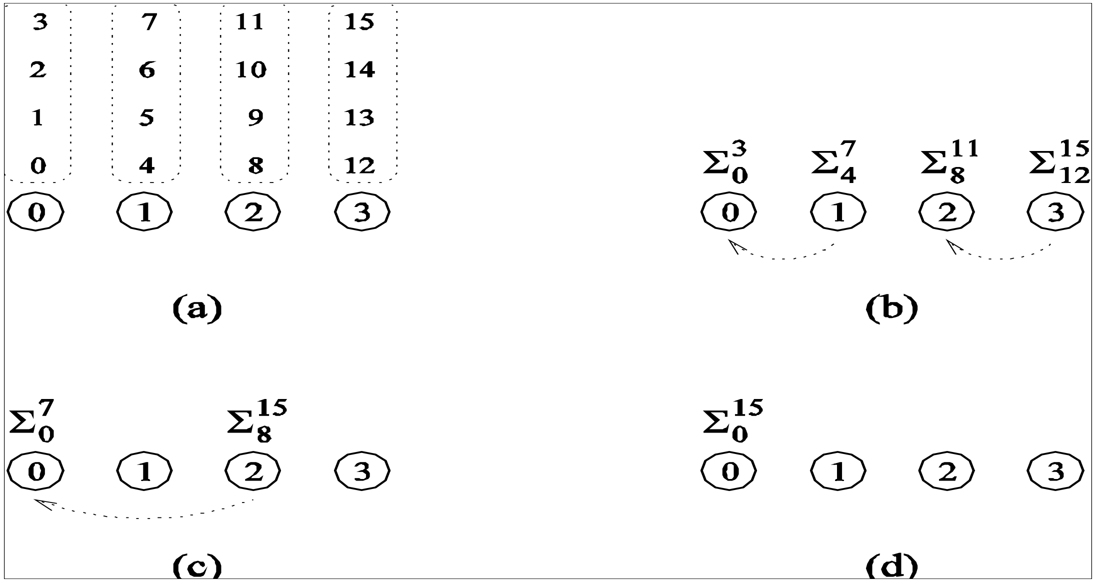
- The overall parallel execution time of this parallel system is $\Theta((n / p) \log p)$.

- The cost is $\Theta(n \log p)$, which is asymptotically higher than the $\Theta(n)$ cost of adding n numbers sequentially. Therefore, the parallel system is not cost-optimal.

Building Granularity: Example (continued)

Can we build granularity in the example in a cost-optimal fashion?

- Each processing element locally adds its ***n*** / ***p*** numbers in time Θ(***n*** / ***p***).
- The ***p*** partial sums on ***p*** processing elements can be added in time Θ(***n*** /***p***).



A cost-optimal way of computing the sum of 16 numbers using four processing elements.

Building Granularity: Example (continued)

- The parallel runtime of this algorithm is

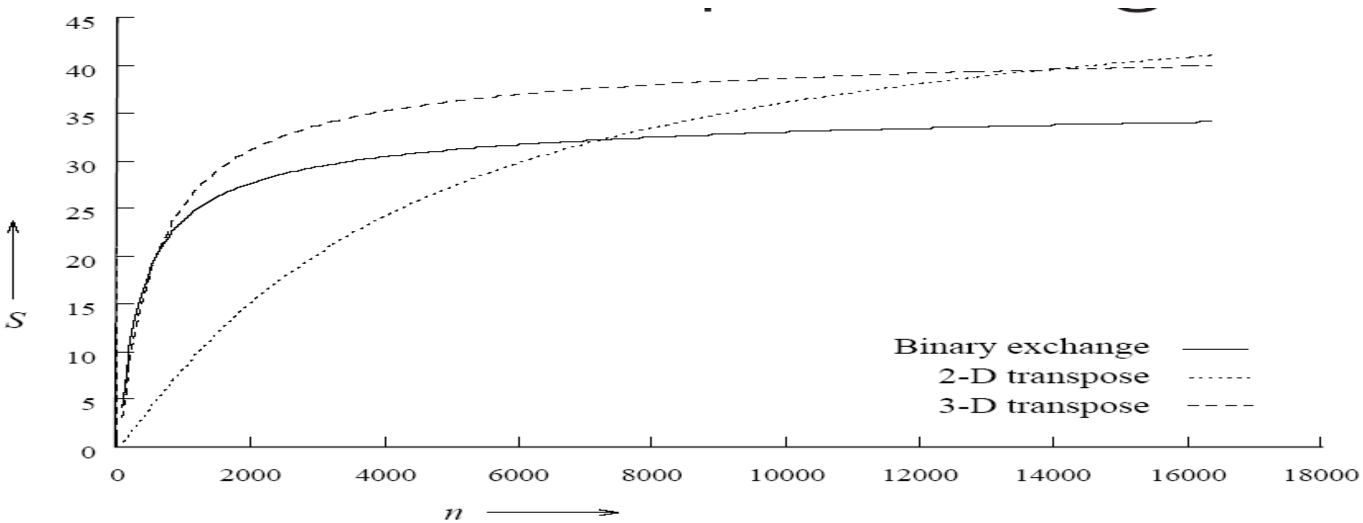
$$T_P = \Theta(n/p + \log p), \tag{3}$$

- The cost is $n = \Omega(p \log p)$
- This is cost-optimal, so long as $\Theta(n + p \log p)$!

Scalability of Parallel Systems

How do we extrapolate performance from small problems and small systems to larger problems on larger configurations?

Consider three parallel algorithms for computing an ***n***-point Fast Fourier Transform (FFT) on 64 processing elements.



A co 2-D transpose and 3-D transpose algorithms on 64 processing elements with $\boldsymbol{t_c}$ = 2, $\boldsymbol{t_w}$ = 4, $\boldsymbol{t_s}$ = 25, and $\boldsymbol{t_h}$ = 2.

Clearly, it is difficult to infer scaling characteristics from observations on small datasets on small machines.

Scaling Characteristics of Parallel Programs

- The efficiency of a parallel program can be written as:

$$E = \frac{S}{p} = \frac{T_S}{pT_P}$$

or

$$E = \frac{1}{1 + \frac{T_o}{T_S}}. \qquad (4)$$

- The total overhead function ***T***

- The total overhead function $\boldsymbol{T_o}$ is an increasing function of ***p*** .

Scaling Characteristics of Parallel Programs

- For a given problem size (i.e., the value of $\boldsymbol{T_S}$ remains constant), as we increase the number of processing elements, $\boldsymbol{T_o}$ increases.

- The overall efficiency of the parallel program goes down. This is the case for all parallel programs.

Scaling Characteristics of Parallel Programs: Example

- Consider the problem of adding   numbers on   processing elements.

- We have seen that:

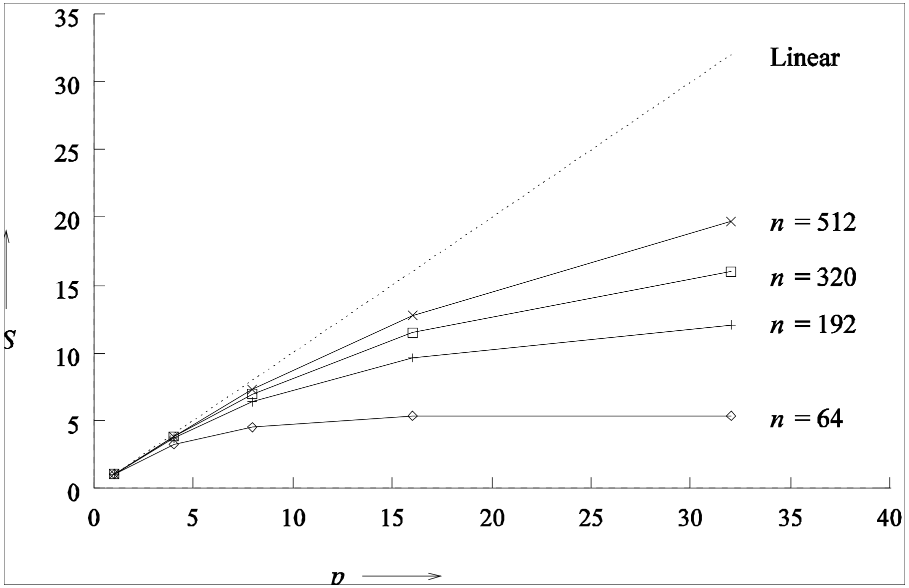
$$T_P = \frac{n}{p} + 2\log p \tag{5}$$

$$S = \frac{n}{\frac{n}{p} + 2\log p} \tag{6}$$

$$E = \frac{1}{1 + \frac{2p\log p}{n}} \tag{7}$$

Scaling Characteristics of Parallel Programs: Example (continued)

Plotting the speedup for various input sizes gives us:



Speedup versus the number of processing elements for adding a list of numbers.

Speedup tends to saturate and efficiency drops.

Scaling Characteristics of Parallel Programs

- Total overhead function $\boldsymbol{T_o}$ is a function of both problem size $\boldsymbol{T_s}$ and the number of processing elements $\boldsymbol{p}$.

- In many cases, $\boldsymbol{T_o}$ grows sub-linearly with respect to $\boldsymbol{T_s}$.

- In such cases, the efficiency increases if the problem size is increased keeping the number of processing elements constant.

- For such systems, we can simultaneously increase the problem size and number of processors to keep efficiency constant.

- We call such systems *scalable* parallel systems.

Scaling Characteristics of Parallel Programs

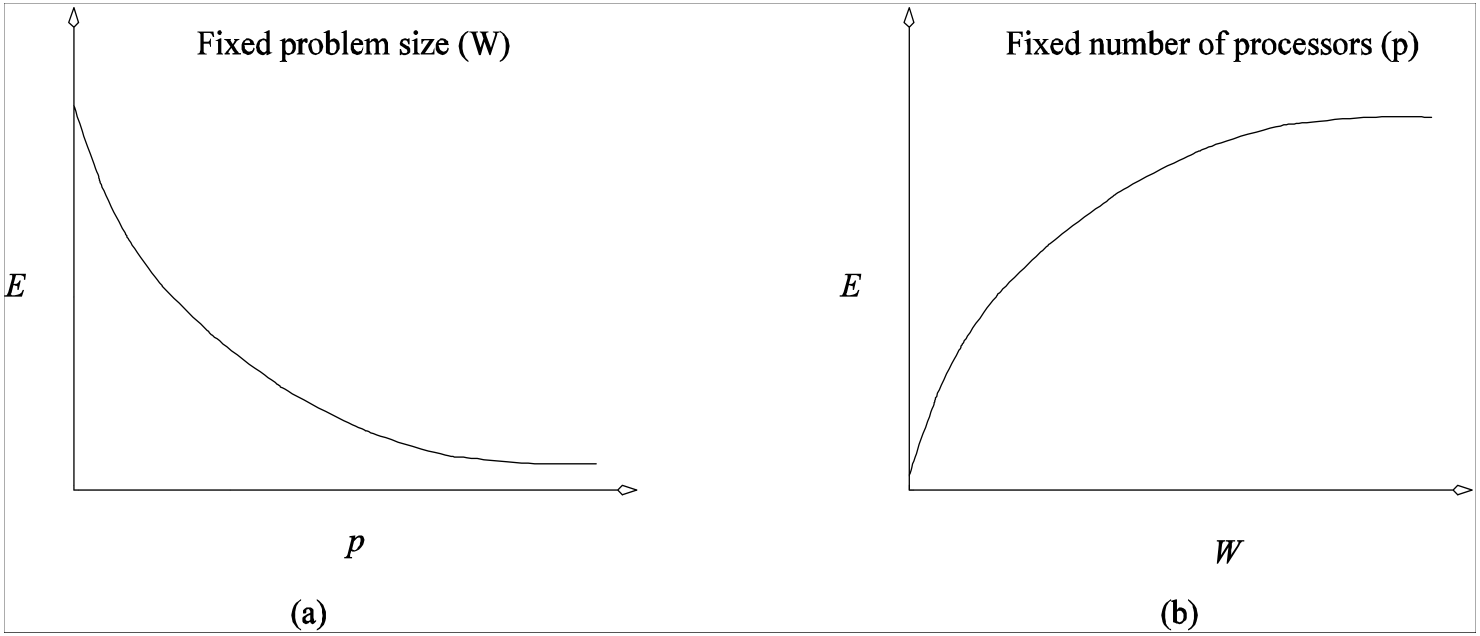
- Recall that cost-optimal parallel systems have an efficiency of Θ(1).
- Scalability and cost-optimality are therefore related.
- A scalable parallel system can always be made cost-optimal if the number of processing elements and the size of the computation are chosen appropriately.

Isoefficiency Metric of Scalability

- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.

- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

Isoefficiency Metric of Scalability



Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

Isoefficiency Metric of Scalability

- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?

- This rate determines the scalability of the system. The slower this rate, the better.

- Before we formalize this rate, we define the problem size ***W*** as the asymptotic number of operations associated with the best serial algorithm to solve the problem.

Isoefficiency Metric of Scalability

- We can write parallel runtime as:

$$T_P = \frac{W + T_o(W,p)}{p} \tag{8}$$

- The resulting expression for speedup is

$$\begin{aligned} S &= \frac{W}{T_P} \\ &= \frac{Wp}{W + T_o(W,p)}. \end{aligned} \tag{9}$$

- Finally, we write the expression for efficiency as

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W,p)} \\ &= \frac{1}{1 + T_o(W,p)/W}. \end{aligned}$$

Isoefficiency Metric of Scalability

- For scalable parallel systems, efficiency can be maintained at a fixed value (between 0 and 1) if the ratio T_o / W is maintained at a constant value.
- For a desired value E of efficiency,

$$\begin{aligned} E &= \frac{1}{1 + T_o(W,p)/W}, \\ \frac{T_o(W,p)}{W} &= \frac{1-E}{E}, \\ W &= \frac{E}{1-E} T_o(W,p). \end{aligned} \qquad (11)$$

- If $K = E / (1 - E)$ is a constant depending on the efficiency to be maintained, since T_o is a function of W and p, we have

$$W = K T_o(W,p). \qquad (12)$$

Isoefficiency Metric of Scalability

- The problem size ***W*** can usually be obtained as a function of ***p*** by algebraic manipulations to keep efficiency constant.

- This function is called the *isoefficiency function*.

- This function determines the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processing elements.

Isoefficiency Metric: Example

- The overhead function for the problem of adding ***n*** numbers on ***p*** processing elements is approximately ***2p* log *p*** .
- Substituting ***T_o*** by ***2p* log *p*** , we get

$$W = K2p\log p. \tag{13}$$

Thus, the asymptotic isoefficiency function for this parallel system is $\Theta(p\log p)$.

- If the number of processing elements is increased from ***p*** to ***p'***, the problem size (in this case, n) must be increased by a factor of (***p'* log *p'***) / (***p* log *p***) to get the same efficiency as on ***p*** processing elements.

Isoefficiency Metric: Example

Consider a more complex example where $T_o = p^{3/2} + p^{3/4}W^{3/4}$

- Using only the first term of $\boldsymbol{T_o}$ in Equation 12, we get

$$W = Kp^{3/2}. \tag{14}$$

- Using only the second term, Equation 12 yields the following relation between ***W*** and ***p***:

$$\begin{aligned} W &= Kp^{3/4}W^{3/4} \\ W^{1/4} &= Kp^{3/4} \\ W &= K^4p^3 \end{aligned} \tag{15}$$

- The larger of these two asymptotic rates determines the isoefficiency. This is given by $\Theta(p^3)$

Cost-Optimality and the Isoefficiency Function

- A parallel system is cost-optimal if and only if

$$pT_P = \Theta(W). \quad (16)$$

- From this, we have:

$$\begin{aligned} W + T_o(W,p) &= \Theta(W) \\ T_o(W,p) &= O(W) \quad (17) \\ W &= \Omega(T_o(W,p)) \quad (18) \end{aligned}$$

- If we have an isoefficiency function ***f***(***p***), then it follows that the relation ***W*** = Ω(***f***(***p***)) must be satisfied to ensure the cost-optimality of a parallel system as it is scaled up.

Lower Bound on the Isoefficiency Function

- For a problem consisting of $\boldsymbol{W}$ units of work, no more than $\boldsymbol{W}$ processing elements can be used cost-optimally.

- The problem size must increase at least as fast as $\Theta(\boldsymbol{p})$ to maintain fixed efficiency; hence, $\Omega(\boldsymbol{p})$ is the asymptotic lower bound on the isoefficiency function.

Degree of Concurrency and the Isoefficiency Function

- The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its *degree of concurrency*.

- If ***C***(***W***) is the degree of concurrency of a parallel algorithm, then for a problem of size ***W***, no more than ***C***(***W***) processing elements can be employed effectively.

Degree of Concurrency and the Isoefficiency Function: Example

Consider solving a system of equations in variables by using Gaussian elimination ($\boldsymbol{W} = \Theta(\boldsymbol{n}^3)$)

- The $\boldsymbol{n}$ variables must be eliminated one after the other, and eliminating each variable requires $\Theta(\boldsymbol{n}^2)$ computations.
- At most $\Theta(\boldsymbol{n}^2)$ processing elements can be kept busy at any time.
- Since $\boldsymbol{W} = \Theta(\boldsymbol{n}^3)$ for this problem, the degree of concurrency $\boldsymbol{C}(\boldsymbol{W})$ is $\Theta(\boldsymbol{W}^{2/3})$.
- Given $\boldsymbol{p}$ processing elements, the problem size should be at least $\Omega(\boldsymbol{p}^{3/2})$ to use them all.

Minimum Execution Time and Minimum Cost-Optimal Execution Time

Often, we are interested in the minimum time to solution.

- We can determine the minimum parallel runtime T_P^{min} for a given W by differentiating the expression for T_P w.r.t. p and equating it to zero.

$$\frac{\mathrm{d}}{\mathrm{d}p} T_P = 0 \qquad (19)$$

- If p_0 is the value of p as determined by this equation, $T_P(p_0)$ is the minimum parallel time.

Minimum Execution Time: Example

Consider the minimum execution time for adding ***n*** numbers.

$$T_P = \frac{n}{p} + 2\log p. \quad (20)$$

Setting the derivative w.r.t. ***p*** to zero, we have ***p = n/ 2*** . The corresponding runtime is

$$T_P^{min} = 2\log n. \quad (21)$$

(One may verify that this is indeed a min by verifying that the second derivative is positive).

Note that at this point, the formulation is not cost-optimal.

Minimum Cost-Optimal Parallel Time

- Let $T_P^{cost\_opt}$ be the minimum cost-optimal parallel time.

- If the isoefficiency function of a parallel system is $\Theta(f(p))$, then a problem of size W can be solved cost-optimally if and only if
$W = \Omega(f(p))$.

- In other words, for cost optimality, $p = O(f^{-1}(W))$.

- For cost-optimal systems, $T_P = \Theta(W/p)$, therefore,

$$T_P^{cost\_opt} = \Omega\left(\frac{W}{f^{-1}(W)}\right). \qquad (22)$$

Minimum Cost-Optimal Parallel Time: Example

Consider the problem of adding ***n*** numbers.

- The isoefficiency function ***f***(***p***) of this parallel system is Θ(***p* log *p***).
- From this, we have ***p*** ≈ ***n*** **/log *n*** .
- At this processor count, the parallel runtime is:

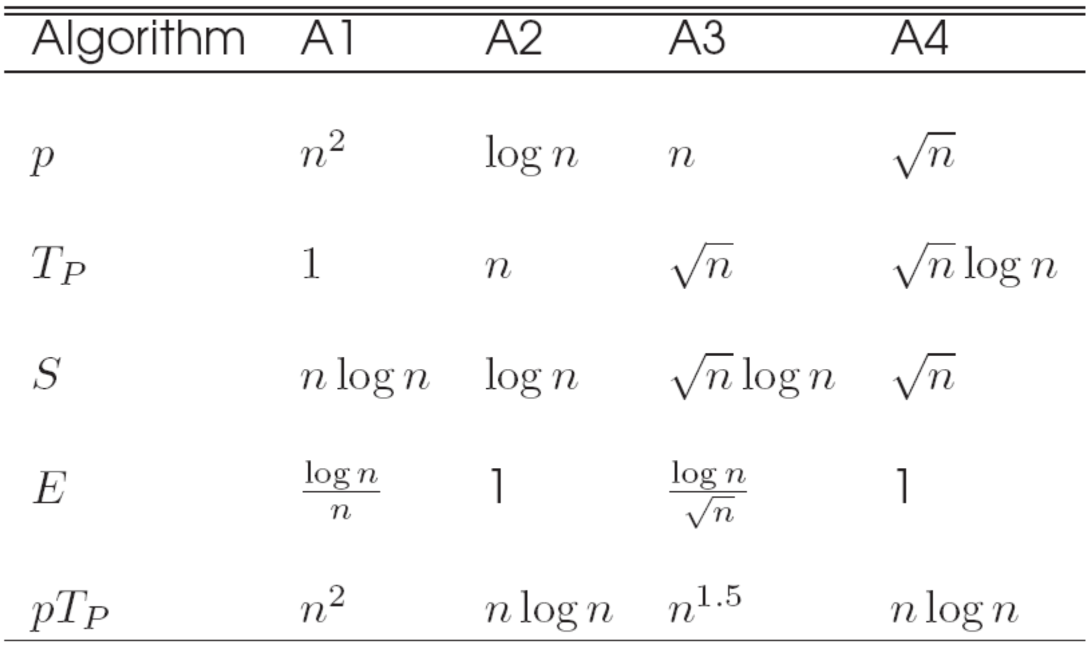
$$
\begin{aligned}
T_P^{cost\_opt} &= \log n + \log\left(\frac{n}{\log n}\right) \\
&= 2\log n - \log\log n. \qquad (23)
\end{aligned}
$$

- Note that both $\boldsymbol{T_P}^{min}$ and $\boldsymbol{T_P}^{cost\_opt}$ for adding ***n*** numbers are Θ(**log *n***). This may not always be the case.

Asymptotic Analysis of Parallel Programs

Consider the problem of sorting a list of ***n*** numbers. The fastest serial programs for this problem run in time Θ(***n*** **log** ***n***). Consider four parallel algorithms, A1, A2, A3, and A4 as follows:

Comparison of four different algorithms for sorting a given list of numbers. The table shows number of processing elements, parallel runtime, speedup, efficiency and the ***pT_P*** product.



Algorithm	A1	A2	A3	A4
p	n^2	$\log n$	n	$\sqrt{n}$
T_P	1	n	$\sqrt{n}$	$\sqrt{n}\log n$
S	$n\log n$	$\log n$	$\sqrt{n}\log n$	$\sqrt{n}$
E	$\frac{\log n}{n}$	1	$\frac{\log n}{\sqrt{n}}$	1
pT_P	n^2	$n\log n$	$n^{1.5}$	$n\log n$

Asymptotic Analysis of Parallel Programs

- If the metric is speed, algorithm A1 is the best, followed by A3, A4, and A2 (in order of increasing T_P).

- In terms of efficiency, A2 and A4 are the best, followed by A3 and A1.

- In terms of cost, algorithms A2 and A4 are cost optimal, A1 and A3 are not.

- It is important to identify the objectives of analysis and to use appropriate metrics!

Other Scalability Metrics

- A number of other metrics have been proposed, dictated by specific needs of applications.

- For real-time applications, the objective is to scale up a system to accomplish a task in a specified time bound.

- In memory constrained environments, metrics operate at the limit of memory and estimate performance under this problem growth rate.

Other Scalability Metrics: Scaled Speedup

- Speedup obtained when the problem size is increased linearly with the number of processing elements.

- If scaled speedup is close to linear, the system is considered scalable.

- If the isoefficiency is near linear, scaled speedup curve is close to linear as well.

- If the aggregate memory grows linearly in ***p***, scaled speedup increases problem size to fill memory.

- Alternately, the size of the problem is increased subject to an upper-bound on parallel execution time.

Scaled Speedup: Example

- The serial runtime of multiplying a matrix of dimension ***n x n*** with a vector is $t_c n^2$.

- For a given parallel algorithm,

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2 t_w \frac{n^2}{\sqrt{p}}} \tag{24}$$

- Total memory requirement of this algorithm is $\Theta(n^2)$.

Scaled Speedup: Example (continued)

Consider the case of memory-constrained scaling.

- We have m= Θ(***n***^{2}) = Θ(***p***).
- Memory constrained scaled speedup is given by

$$S' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + t_w \sqrt{c \times p}}$$

or $S' = O(\sqrt{p})$

- This is not a particularly scalable system

Scaled Speedup: Example (continued)

Consider the case of time-constrained scaling.

- We have $T_P = O(n^2)$.
- Since this is constrained to be constant, $n^2 = O(p)$.
- Note that in this case, time-constrained speedup is identical to memory constrained speedup.

Scaled Speedup: Example

- The serial runtime of multiplying two matrices of dimension ***n x n*** is $t_c n^3$.

- The parallel runtime of a given algorithm is:

$$T_P = t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$$

- The speedup ***S*** is given by:

$$S = \frac{t_c n^3}{t_c \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}} \qquad (25)$$

Scaled Speedup: Example (continued)

Consider memory-constrained scaled speedup.

- We have memory complexity m= Θ(***n***^{2}) = Θ(***p***), or ***n***^{2}=***c x p .***

- At this growth rate, scaled speedup ***S'*** is given by:

$$S' = \frac{t_c(c \times p)^{1.5}}{t_c\frac{(c\times p)^{1.5}}{p} + t_s \log p + 2t_w\frac{c\times p}{\sqrt{p}}} = O(p)$$

- Note that this is scalable.

Scaled Speedup: Example (continued)

Consider time-constrained scaled speedup.

- We have T_P = $O(1) = O(n^3 / p)$, or $n^3 = c \times p$.
- Time-constrained speedup ***S”*** is given by:

$$S'' = \frac{t_c c \times p}{t_c \frac{c \times p}{p} + t_s \log p + 2 t_w \frac{(c \times p)^{2/3}}{\sqrt{p}}} = O(p^{5/6})$$

- Memory constrained scaling yields better performance.

Serial Fraction *f*

- If the serial runtime of a computation can be divided into a totally parallel and a totally serial component, we have:

$$W = T_{ser} + T_{par}.$$

- From this, we have,

$$T_P = T_{ser} + \frac{T_{par}}{p}.$$

$$T_P = T_{ser} + \frac{W - T_{ser}}{p} \tag{26}$$

Serial Fraction *f*

- The serial fraction ***f*** of a parallel program is defined as:

$$f = \frac{T_{ser}}{W}.$$

- Therefore, we have:

$$T_P = f \times W + \frac{W - f \times W}{p}$$

$$\frac{T_P}{W} = f + \frac{1 - f}{p}$$

Serial Fraction

- Since ***S* = *W* / *T*$_P$** , we have

$$\frac{1}{S} = f + \frac{1-f}{p}.$$

- From this, we have:

$$f = \frac{1/S - 1/p}{1 - 1/p}. \tag{27}$$

- If ***f*** increases with the number of processors, this is an indicator of rising overhead, and thus an indicator of poor scalability.

Serial Fraction: Example

Consider the problem of extimating the serial component of the matrix-vector product.

We have:

$$f = \frac{\frac{t_c \frac{n^2}{p} + t_s \log p + t_w n}{t_c n^2}}{1 - 1/p} \tag{28}$$

or

$$f = \frac{t_s p \log p + t_w n p}{t_c n^2} \times \frac{1}{p-1}$$

$$f \approx \frac{t_s \log p + t_w n}{t_c n^2}$$

Here, the denominator is the serial runtime and the numerator is the overhead.