Lecture 7
Analytical performance analysis

Examples: Speed-up, Efficiency and Cost
Granularity
Scalability

Speed-up, Efficiency, Cost - review

- Speed-up

  $S = Ts/Tp$

- Efficiency

  $E = S/p$

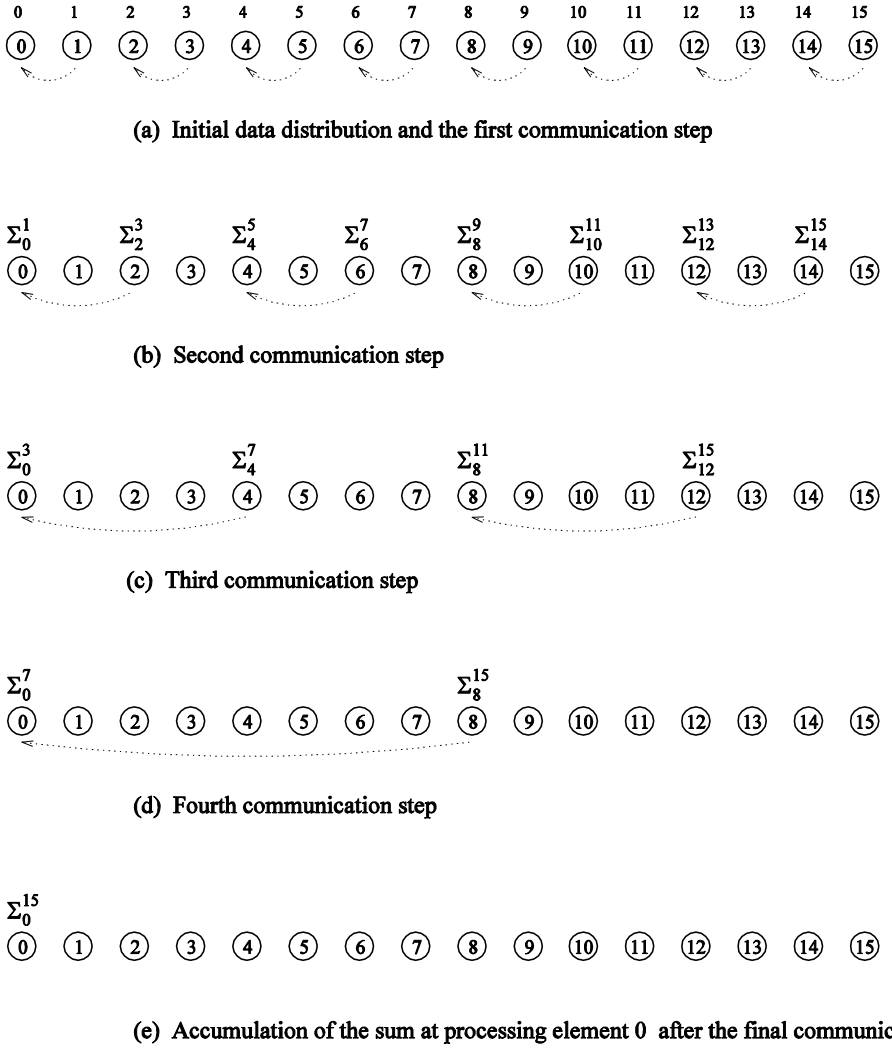
Cost of a Parallel System

- Cost is the product of parallel runtime and the number of processing elements used $C = p \times T_P$
- 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 - $C = \Theta(T_s)$
- Since $E = T_S / p\,T_P$, for cost optimal systems, $E = O(1)$.
- Cost is sometimes referred to as ***Work*** or ***processor-time product***.

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



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

Performance Metrics evaluation:

Adding on a distributed memory systems

=>If an addition takes constant time - t_c and

- communication of a single word takes time $t_s + t_w$,
- $p=n$

$T_P = \log n \, [\, (t_s + t_w) + t_c \,]$

$T_S = n \, t_c$

$S(n) = T_S / T_P = (n / \log n) \, (t_c / (t_s + t_w + t_c))$

$T_P = \Theta (\log n)$

$T_S = \Theta (n)$

- Asymptotic Speedup S is given by $S = \Theta (n / \log n)$

Performance Metrics: Efficiency
Adding numbers Example

- The speedup of adding numbers on processors is given by
- Efficiency is given by :

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

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

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

Cost of a Parallel System: Adding numbers Example

Consider the problem of adding numbers on processors.

- We have, $T_P = \log n$ (for $p = n$).
- The cost of this system is given by $p\,T_P = n \log n$.
- Since the serial runtime of this operation is $\Theta(n)$, 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***.

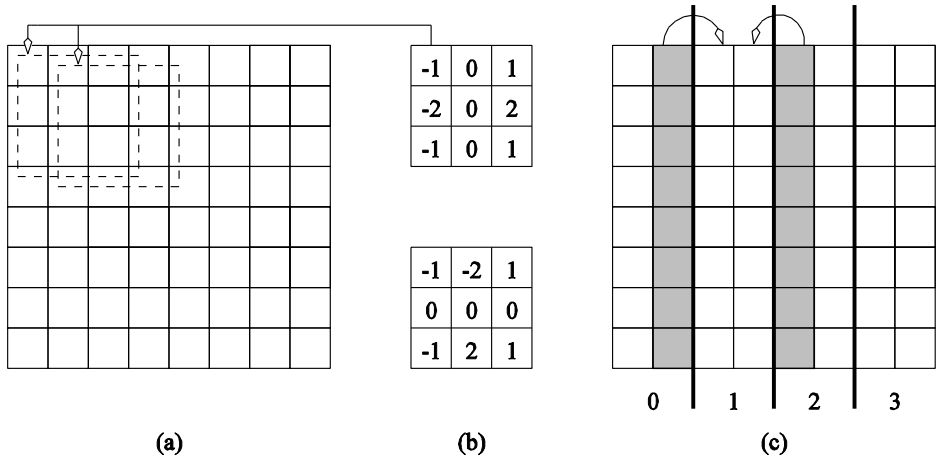
- If $p < n$, assigning the n tasks to p processors gives $T_P = (\log n)^2 (n / p)$.

- The corresponding speedup of this formulation is p / $\log n$.

- For a given **p** this speedup goes down as the problem size n is increased!

Parallel Time, Speedup, and Efficiency Example

Consider the problem of edge-detection in images. The problem requires us to apply a ***3*** x ***3*** template to each pixel. If each multiply-add operation takes time t_c, the serial time for an ***n*** x ***n*** image is given by $T_S = t_c n^2$.



Example of edge detection: (a) an ***8*** x ***8*** image; (b) typical templates for detecting edges; and (c) partitioning of the image across four processors with shaded regions indicating image data that must be communicated from neighboring processors to processor 1.

Parallel Time, Speedup, and Efficiency Example (continued)

- One possible parallelization partitions the image equally into vertical segments, each with n^2 / p pixels.

- The boundary of each segment is $2n$ pixels. This is also the number of pixel values that will have to be communicated.
  - $T^{Communication}_{p}(n) = 2(t_s + t_w n)$.

- Templates may now be applied to all n^2 / p pixels in time
  - $T^{Computation}_{p}(n) = 9\, t_c n^2 / p$.

Parallel Time, Speedup, and Efficiency Example (continued)

- The total time for the algorithm is therefore given by:

$$T_P = 9t_c\frac{n^2}{p} + 2(t_s + t_w n)$$

- The corresponding values of speedup and efficiency are given by:

$$S = \frac{9t_c n^2}{9t_c\frac{n^2}{p} + 2(t_s + t_w n)}$$

and

$$E = \frac{1}{1 + \frac{2p(t_s + t_w n)}{9t_c n^2}}.$$

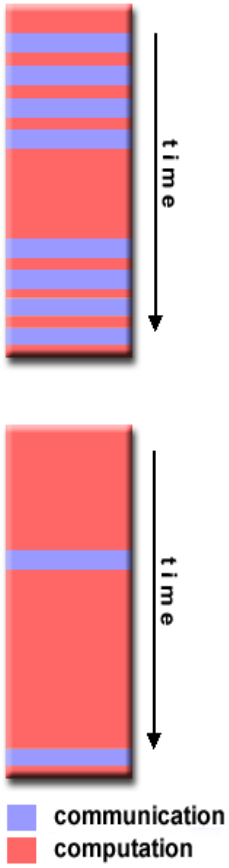
$C = (9\ t_c n^2/p + 2(t_s + t_w n))p = 9\ t_c n^2 + 2p(t_s + t_w n)$

Granularity(in parallel computing)

Granularity measures the amount of computation in relation to communication.

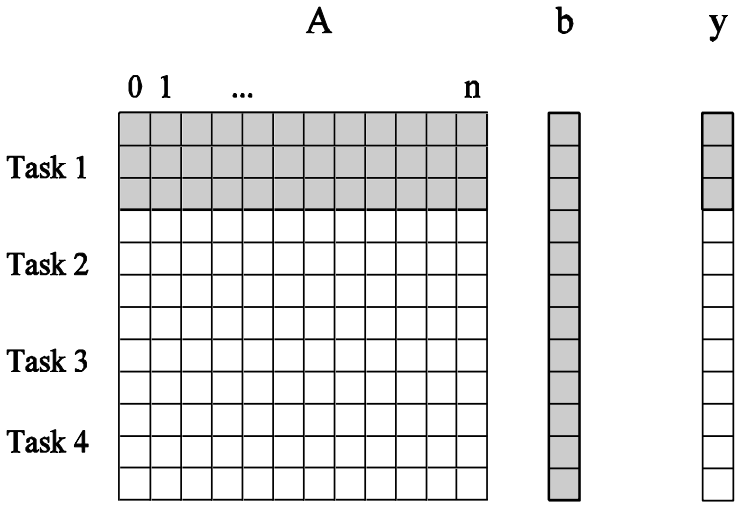
- could be approximated by the ratio of computation to the amount of communication
- it is determined by the minimum size (no of instructions) of a unity of computation work (in which no communication and synchronization take place)

- **Fine-grain Granularity:**
  - Relatively small amounts of computational work are done between communication/synchronization events
  - Low computation to communication /synchronization ratio
  - Facilitates load balancing
  - Implies high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.
- **Coarse-grain Granularity:**
  - Relatively large amounts of computational work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently



Granularity of Task Decompositions

- The number of tasks into which a problem is decomposed **determines(not equal !!!)** its granularity.
- Decomposition into a large number of tasks results in fine-grained decomposition and that into a small number of tasks results in a coarse grained decomposition.



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.

Granularity…cont.

- Periods of computation are typically separated from periods of communication or synchronization events.
- In most cases the overhead associated with communications and synchronization is high relative to computation speed.

Which is Best?

- The finer the granularity is, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication.

- In order to attain the best parallel performance, the best balance between load and communication overhead needs to be found.
  - If the granularity is too fine, the performance can suffer from the increased communication overhead.
  - If the granularity is too coarse, the performance can suffer from load imbalance.

- ***The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.***

Effect of Granularity on Performance

- Often, using fewer processors improves efficiency 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 main reason of 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 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 $\Theta ((n / p) \log p)$.

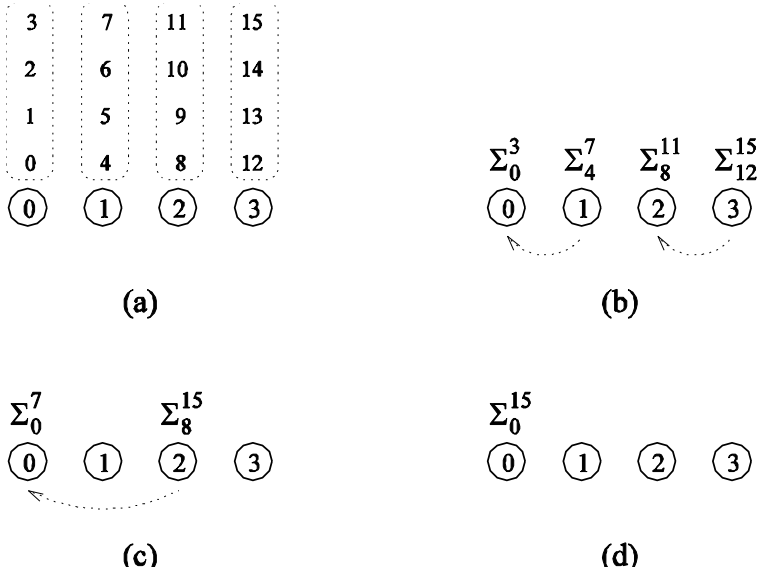
- The cost is $\Theta (n \log p)$, which is asymptotically higher than the $\Theta (n)$ cost of adding n numbers sequentially.

- 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 $\Theta(n / p)$.
- The p partial sums on p processing elements can be added in time $\Theta(n / p)$.



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

$$T_P = \Theta(n/p + \log p),$$

- so long as

$$n = \Omega(p \log p)$$

- The cost is cost-optimal $C(n) = pT_p(n) = \Theta(n + p \log p)$

- $T_s(n) = \Theta(n) = C(n) = pT_p(n) = \Theta(n)$
- $C(n, p) = \Theta(T_s(n))$ the condition for cost optimality
- => this variant is **cost-optimal**

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.
- The table shows number of processing elements, parallel runtime, speedup efficiency and cost.

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 sorting algorithms

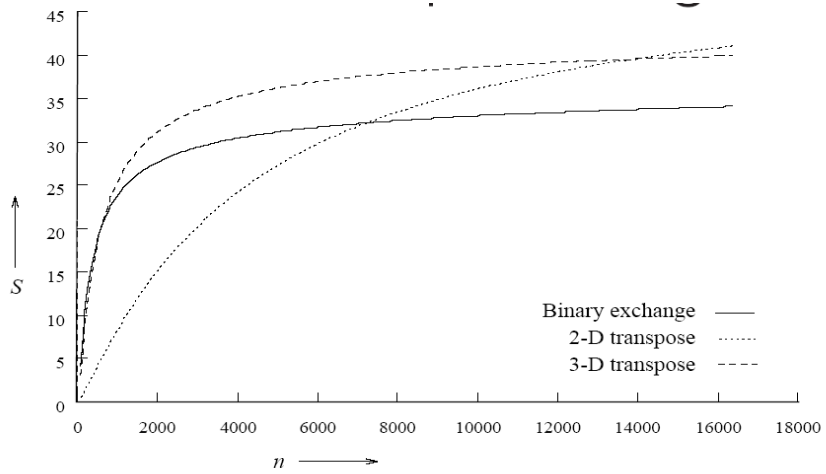
- If the metric is speed, algorithm A1 is the best, followed by A3, A4, and A2 (in order of increasing T_P).
- In terms of efficiency, A2 and A4 are the best, followed by A3 and A1.
- In terms of cost, algorithms A2 and A4 are cost optimal, A1 and A3 are not.

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

Scalability of Parallel Systems

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

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



A comparison of the speedups obtained by the binary-exchange, 2-D transpose and 3-D transpose algorithms on 64 processing elements with $t_c = 2$, $t_w = 4$, $t_s = 25$, and $t_h = 2$.

- 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

$$E = \frac{1}{1 + \frac{T_o}{T_S}}.$$

- The total overhead function T_o is an increasing function of p .
  - $T_o = pT_p - T_s$
  - $pT_p = T_s + T_o$

Scaling Characteristics of Parallel Programs

- For a given problem size (i.e., the value of T_S remains constant), as we increase the number of processing elements, 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 n numbers on p processing elements.

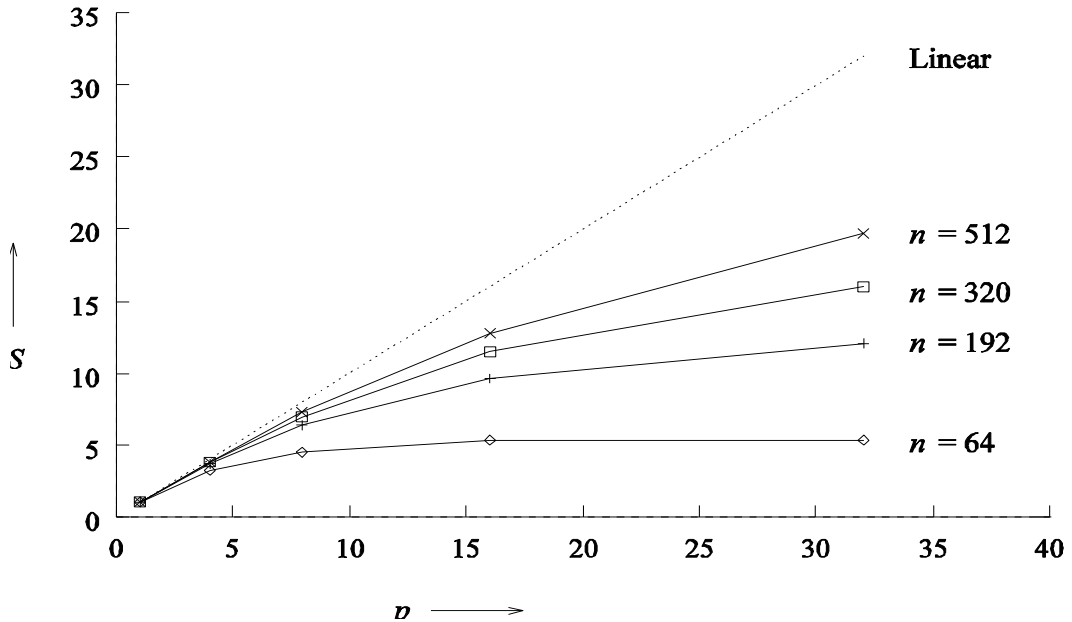
$$T_p = n/p + 2 \log p$$

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

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

Scaling Characteristics of Parallel Programs
Example: adding a list of numbers

- Speedup for various input sizes:



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

- Speedup tends to saturate, and efficiency drops as a consequence of Amdahl's law

Scaling Characteristics of Parallel Programs

- Total overhead function T_o is a function of both problem size T_s and the number of processing elements p.

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

- In many cases, T_o grows sublinearly with respect to T_s.

$$E = \frac{1}{1 + \frac{T_o}{T_S}}.$$

- 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

- 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.

Metrics of Scalability

Basic:

- Scalability (scalab(n)) the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

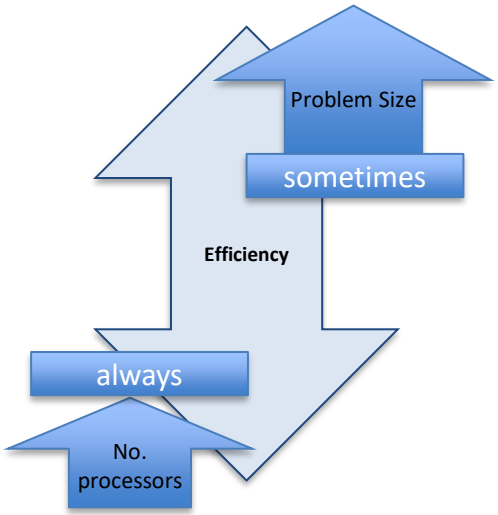
$$scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

- It measures how efficient is the parallel implementation in achieving better performances on larger parallelism degrees.
- As for the speedup, the asymptote is linear (f(n) = n) proof – similar to that for speed-up.
- However, scalability does not provide any comparison with the “best" sequential computation.

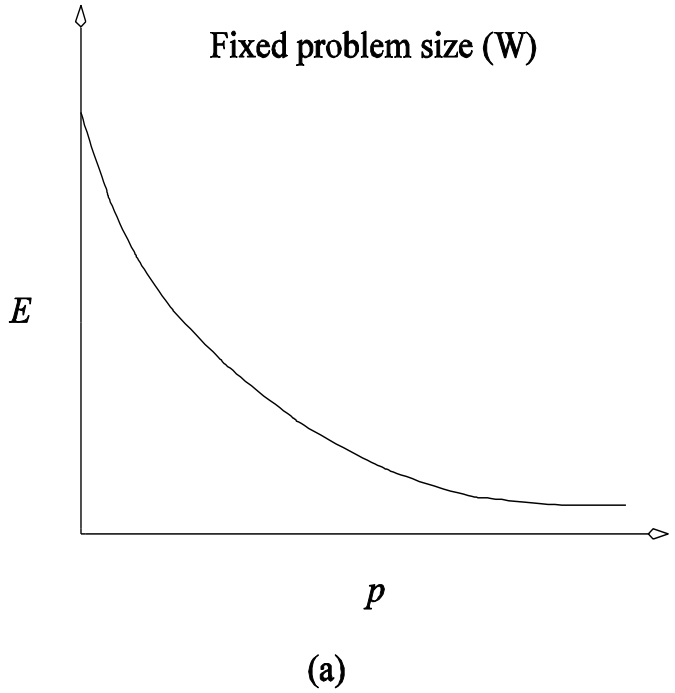
Iso-efficiency Metric of Scalability

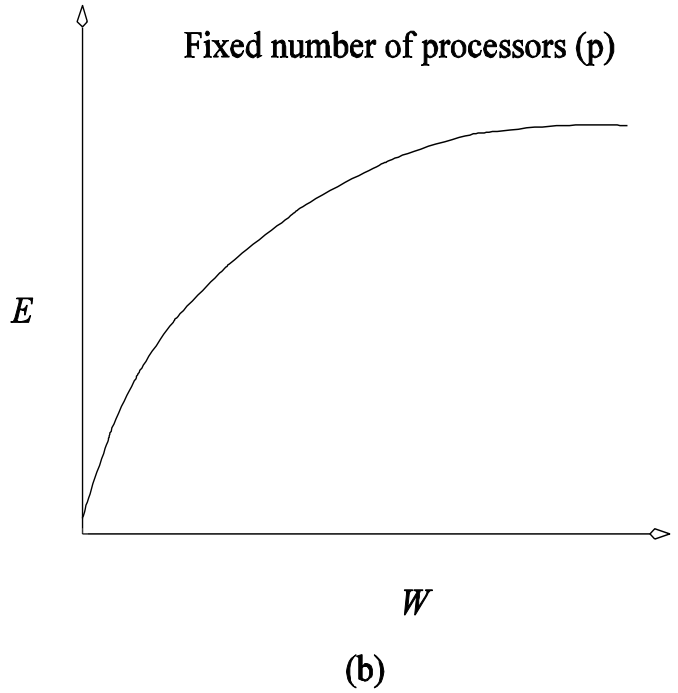
Recall that:

- For all systems, as we increase the number of processing elements, the overall efficiency of the parallel system goes down, if we maintain a given problem size constant.
- For some systems, the efficiency of a parallel system increases if *the problem size(work) 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 be increased 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*** *(work)* 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:

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

- The resulting expression for speedup is

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

- 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,

$$E = \frac{1}{1 + T_o(W,p)/W},$$

$$\frac{T_o(W,p)}{W} = \frac{1-E}{E},$$

$$W = \frac{E}{1-E} T_o(W,p).$$

- 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

$$W = K T_o(W,p).$$

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 = K*2*p* log p

  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.

Example:

- if ***p=4, p'=8 => W' = (8/4)(3/2)W => W'= 3W***

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 T_o in $W = KT_o(W, p).$

$W = K p^{3/2}$

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

$$W = Kp^{3/4}W^{3/4}$$

$$W^{1/4} = Kp^{3/4}$$

$$W = K^4p^3$$

The larger of these two asymptotic rates determines the asymptotic isoefficiency.

This is given by $\Theta(p^3)$

Cost-Optimality and the Isoefficiency Function

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

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

- From this, we have:

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

- 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 ***W*** units of work, no more than ***W*** processing elements can be used cost-optimally.

- The problem size must increase at least as fast as Θ(***p***) to maintain fixed efficiency; hence, Ω(***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 $\Theta(\boldsymbol{W}^{2/3})$ (n- expressed in function of W) .
- 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.

(find the number optimal number of processors)

- 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.

$$\frac{d}{dp} T_P = 0$$

- 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.

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

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

$$T_P^{min} = 2\log n.$$

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

Minimum Cost-Optimal Parallel Time: Example

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

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

$$T_P^{cost\_opt} = \log n + \log\left(\frac{n}{\log n}\right)$$

$$= 2\log n - \log\log n.$$

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

Other scalability metrics

Serial Fraction *f – another metric for scalability*

- 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,

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

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

Serial Fraction ***f*** *(… Amdahl law…)*

- 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:

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

- 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 estimating the serial component of the matrix-vector product (on a hypercube).

We have:

$$f = \frac{\frac{t_c n^2}{p} + t_s \log p + t_w n}{t_c n^2} \Big/ (1 - 1/p)$$

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.

Isospeed scalability metric

X.H. Sun and D. Rover, "Scalability of Parallel Algorithm–Machine Combinations", *IEEE Transaction on Parallel Distributed Systems*, Vol. 5, pp.599–613, 1994.

- An algorithm-machine combination is defined to be scalable if the achieved average unit speed of the algorithm on the given machine can remain constant with increasing number of processors, provided the problem size can be increased with the system size (the average unit speed (speed per processor) is defined as the system's achieved speed divided by the number of processors).

- The isospeed scalability function is $$\Psi(p, p') = \frac{p' W}{p W'}$$

where p and p' are the initial and scaled number of processors,
and W and W' are the initial and scaled work (problem size) respectively.

- The isospeed scalability works well in homogeneous environment.

isospeed-e scalability metric

X.H. Sun, Y. Chen and M. Wu, "Scalability of Heterogeneous Computing", *Proceedings of 34th International Conference on Parallel Processing*, pp.557-564, 2005.

- In *isospeed-e scalability* metric a new concept of *marked speed* is introduced to describe the combined computing power of a general parallel computing system.

- The marked speed of a computing node is the (benchmarked) sustained speed of that node. It represents the computational capability of that node. It can be calculated based on hardware peak performance, which in general is higher than actually delivered performance. In practice, computation intensive benchmarks can be used to measure the marked speed of each node.

- Once the marked speed of a computing node is measured, it is used as a constant parameter, like CPU frequency or memory access latency, in all experimental analyses.

- The marked speed of a computing system is defined as the sum of the marked speed of each node that composes the computing system. Let *Ci* denote the marked speed of node *i*. In a heterogeneous environment, *Ci* may be different from each other due to the heterogeneity. In homogeneous environment, all *Ci* are the same. Let *C* stand for the marked speed of a computing system.