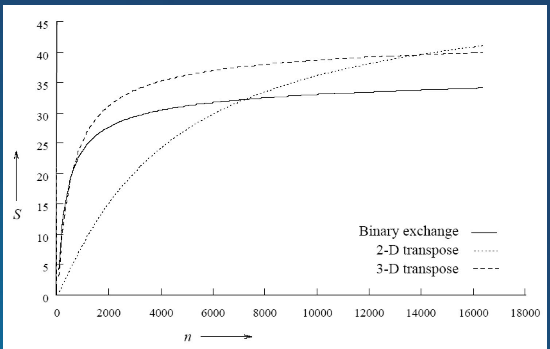
Performance Evaluation

Frédéric Desprez
INRIA



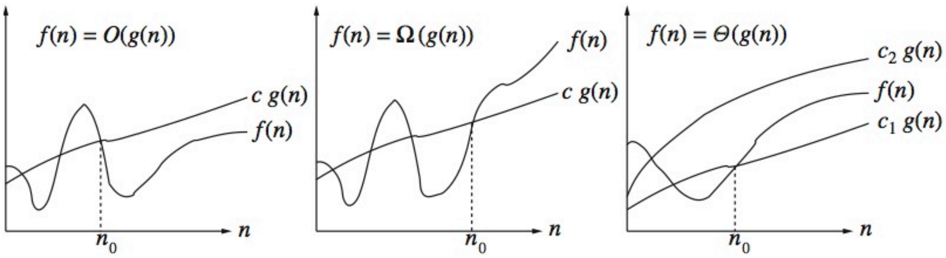
Some references

- **Parallel Programming – For Multicore and Cluster System,** T. Rauber, G. Rünger

- **Introduction to parallel Computing, 2nd Edition**, A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison Wesley

Orders of magnitude

- The order of magnitude of a monotonically increasing function $f = f(x)$ can be expressed in different ways
- **The most well known**
  - The ***O*(*x*)** notation which gives an **upper bound on the order of the function** (i.e. its rate of variation or growth);
    - There exists a positive constant c and n_0 integer such that for any $n >= n_0 : 0 <= f(n) <= c\ g(n)$
    - We then guarantee that $f(n)$ increases, at most, as fast as $O(n)$ from $n > n_0$.
    - **Upper bound** of complexity
  - The **$\Omega(n)$** notation which gives a **lower bound on the order of the function**;
    - There exists a positive constant c and n_0 integer such that for any $n >= n_0$: $0 <= c\ g(n) <= f(n)$
    - We then guarantee that $f(n)$ increases, at least, as fast as $\Omega(n)$ from $n > n_0$.
    - **Lower bound** of complexity
  - The **Θ(*n*)** which is a **combination of the first two** and which can therefore give a more precise idea of the order of magnitude of the function;
    - There are positive constants c_1 and c_2 and n_0 integer, such that for any $n >= n_0 : 0 <= c_1\ g(n) <= f(n) <= c_2\ g(n)$
    - We then guarantee that $f(n)$ grows as fast as $\Theta(n)$ from $n > n_0$.
    - **Equivalence** in complexity



Measuring time

Before parallelizing a program, one must be able to know which part of a program takes the most time in computation

- **Three types of time to consider**
  - **Wall time**
    - The time spent executing a program: the time spent between the beginning of the execution and the end
  - **User time**
    - The time really used by the program
    - It can be much lower than the wall time if the program has to wait a lot, for example for system calls or data exchanges
    - This lost time can give indications for optimizations
  - **System time**
    - Time not used by the program itself but by the operating system (memory allocation, process management, disk access, ...)
    - We try to keep it minimal

Measuring time, contd.

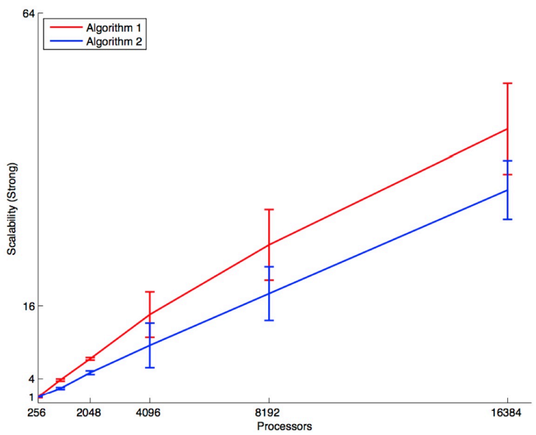
- Unix time command: `time ./executable`
  - Output example

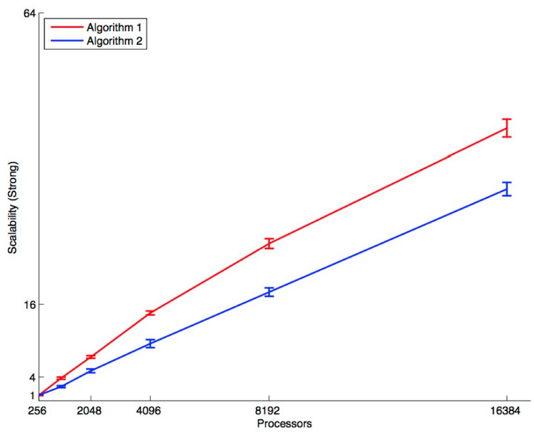
```
real 3m13.535s
user 3m11.298s
sys 0m1.915s
```

  - Measures the total time of the program
- For performance analysis, it is necessary to know the execution time of certain parts of the program
  - Methods dependent on programming languages or operating systems
  - MPI: MPI_Wtime(), OpenMP: omp_get_wtime()
    - Give the wall time between two function calls
- Application profiling
  - If proper compilation, use gprof (`gprof executable > prof.txt`)
  - List of all functions with their execution time, their total time percentage, number of calls
  - Call tree
- Software timers
  - PAPI

Good Measurement Practices

- Choice of number of processors
  - Depending on available resources
  - Beware of physical topology
- Pay attention to the resolution of the clock
- Repeat experiments to understand variability
  - Shared resources (processors, network)
  - Placing jobs / threads on potentially different processors / cores
- Confidence Interval





Need for analytical models of parallel programs

- A sequential program can be evaluated according to its given execution time according to the size of its input data
- A parallel program has its time that depends on other elements
  - Number of processors used
  - Their relative speed
  - The speed of communication between them

  $\Rightarrow$ A parallel program can not be evaluated independently of these elements
- **Some intuitive measures**
  - The wall time obtained to solve a given problem on a given parallel platform
  - What is the gain obtained in speed with respect to the sequential time: the acceleration (or speedup)

Execution time

- **Sequential execution time (T_s)**
  - It is the time spent between the beginning and the end of an execution on a sequential node
- **The parallel time (T_p)**
  - This is the time between the start of parallel execution and the time the last processor finishes
- **Warning!**
  - To compare, use the same processors!
  - Take the data transfers into account if necessary

Factors Affecting Performance

- The algorithm should be able to be parallelized!
- The volume of data to which it applies must be sufficiently large in relation to the number of processors used
- Additional overhead due to synchronization and memory access conflicts can reduce performance
- Load balancing between processors
- The use of parallel algorithms can increase the complexity of parallel algorithms compared to sequential algorithms
- The distribution of data between multiple memory units can reduce memory contention and improve the locality of the data, which can lead to performance gains

Overhead sources

- **Interactions between processes**
  - A non-trivial parallel algorithm will require interactions between processes during execution (synchronization, intermediate data exchange)
  - Communications are generally the most important sources of performance loss

- **Waiting time**
  - Because of many reasons like
    - A load imbalance,
    - synchronizations,
    - the presence of sequential parts.

Overhead sources

The fastest sequential algorithms for a given problem may prove to be difficult / impossible to parallelize

- Using a parallel algorithm based on a sequential algorithm that is simpler to parallelize (with a high degree of concurrency)
- Example: matrix product using Strassen or Winograd algorithms vs 3 loops

Difference between the number of operations between the best sequential algorithm and the parallel algorithm

- Overhead in number of operations
- But a parallel algorithm based on the best sequential algorithm can still perform more calculations than the sequential algorithm
- Example: Fast Fourier Transform (FFT)
  - In the sequential version, the results of some computations can be reused
  - In the parallel version, generated by different processors (thus performed several times by different processors)

Extra cost

- The extra costs induced by a parallel algorithm are encapsulated in a unique expression called an **extra cost function**
- The total overhead or overhead cost of a parallel system (T_o) is defined as the total time taken by all processors **over** the time required for the **fastest sequential algorithm** on a single processor
- Thus
  - Total time to solve a problem summed up on all processors: pT_p
  - T_s units of this time to do useful work
  - What's left is extra cost

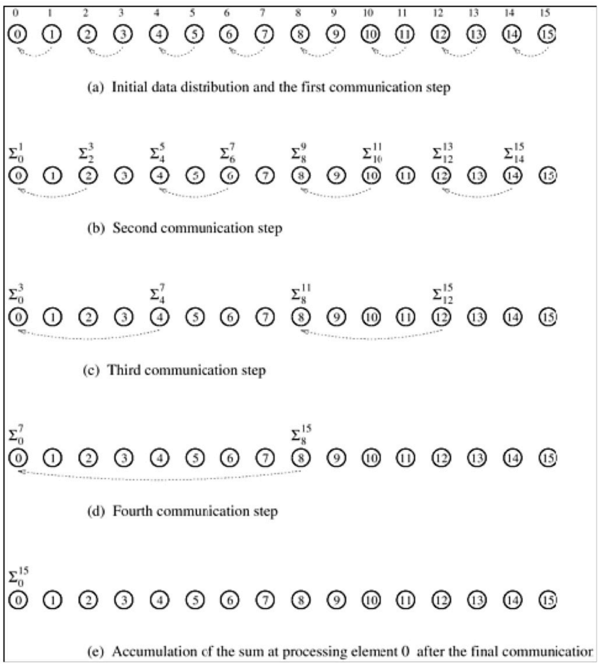
$\Rightarrow$ The extra cost function is given by: $T_o = pT_p - T_s$

Acceleration (*speedup*)

- What **performance gain can be achieved** by parallelizing an application compared to its sequential implementation?
- The speedup is a measure that captures the relative benefit of solving a problem in parallel
- The speedup *S* is the **ratio of time to solve a problem on a single processor over time to solve a problem on a parallel *p* processors machine**
- It generally ranges between 0 and *p*, where *p* is the number of processors
  - Same type of processors between parallel and sequential execution
  - One should (normally) take the best sequential algorithm to solve the same problem
    - Sometimes it is not known or its implementation makes it ineffective
    - Then take the best implementable algorithm

Example: adding n numbers over n processors

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



$Tp = \Theta(\log n)$

$Ts = \Theta(n)$

$S = \Theta(n/\log n)$

Example: integer sort

- Bubble sort parallelization
- We find
  - A sequential version for 105 entries: 150 s
  - Quick sort: 30 s
  - A parallel version (*odd-even sort* over 4 processors): 40 s

- By taking the same two versions, we get a speedup of

$$150/40 = 3.75$$

- But with the best algorithm, we get

$$30/40 = 0.75$$

Theoretically S<p

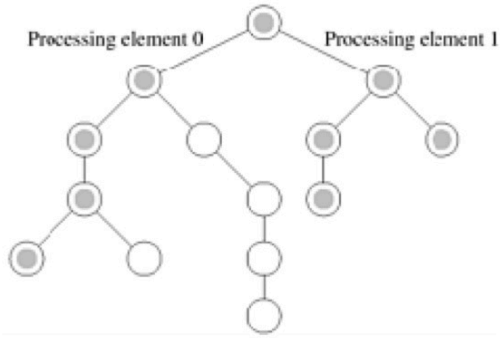
- If the best sequential algorithm takes T_s units of time to solve a given problem on a single processor, then an speedup of p can be obtained for p processors if none of the processors takes more than T_s/p

- Suppose that $S>p$
  - Possible only if each processor spends less than T_s/p units of time to solve the problem
  - A single processor could emulate p processors and solve the problem in less than T_s units of time
  - Contradiction because the acceleration S is calculated with respect to the best sequential algorithm

Superlinear speedup

- There are sometimes accelerations greater than p

- This happens when
  - The work done by a sequential algorithm is superior to that of its parallel version
  - Exemple: search, algorithms in trees



  - If the data enters the caches for the parallel version
    - The performance of larger memory sizes is less important

Efficiency

- Efficiency measures the fraction of time for which a processor is used in a useful way

$$E = S/p$$

  - An efficient system has an efficiency equal to 1
  - In practice $0 \leq E \leq 1$

Cost

- The cost is equal to the parallel time multiplied by the number of processors used : pT_p
- It reflects the time spent by each processor to solve the problem

- A parallel system is cost optimal if
  - O(solve a problem in parallel) = O(to solve it sequentially)

- As $E = T_s/pT_p$, for cost optimal systems $E = O(1)$

Cost optimality and *n* numbers sum

- Is the parallel version on *n* nodes cost optimal?

- $T_p = log\ n$ for $p = n$
- Cost of this parallel version = $p\ T_p = \Theta\ (n\ log\ n)$
- Sequential time = $\Theta\ (n)$

- The algorithm is not cost optimal
  - $E = \Theta\ (n\ /\ (n\ log\ n)) = \Theta\ (1\ /\ log\ n)$

Impact of non cost-optimality

- Using n processors to sort a vector in $(\log n)^2$
  - Bitonic sort [Batcher, 1968]
- Sequential sort using comparisons: n log n
- Speedup = n log n / $(\log n)^2 = n / \log n$
- Cost = $p\ T_p$ of this algorithm is $n (\log n)^2$
  - Not cost optimal of a factor of $\log n$
- If $p < n$
  - Assign n tasks with p processors: $T_p = n (\log n)^2 / p$
  - Speedup = $n \log n / (n (\log n)^2 / p) = p / \log n$
  - Speedup decreases when size increases for a given p
  - High cost associated with lack of optimality in cost
    - Beware of cost optimality!

Effect of granularity on performance

- **Reduce the size of the parallel platform**
  - Use fewer processors than available
  - Usually improves the efficiency of a parallel system
    - Consider each processor as a virtual processor
    - Map virtual processors to the reduced subsystem of processors

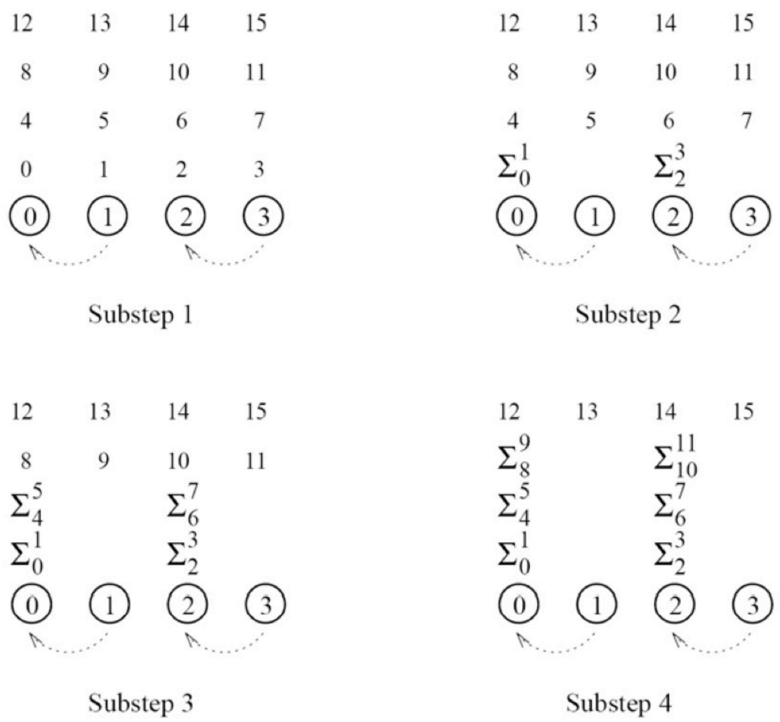
- **Impact**
  - If the number of processors decreases by a factor of n / p
  - The compute volume for each processor increases by one factor of n / p
  - The communication costs depend on what the virtual processors do

Improving granularity: the example of the sum

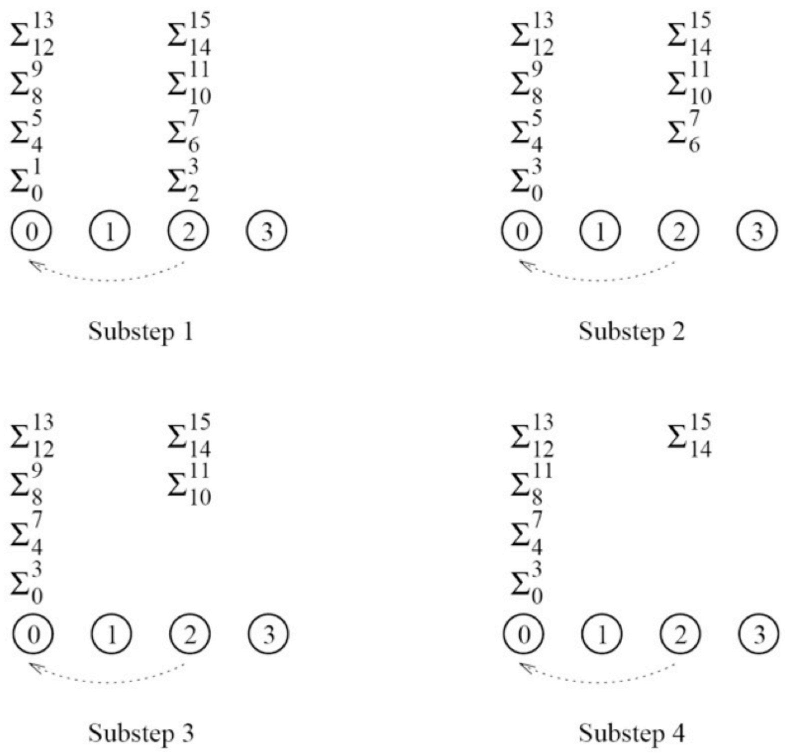
- Adding n numbers on p processors
  - $p < n$
  - n and p are power of 2

- Using an algorithm for n virtual processors
  - Assign to each physical processor n / p virtual processors

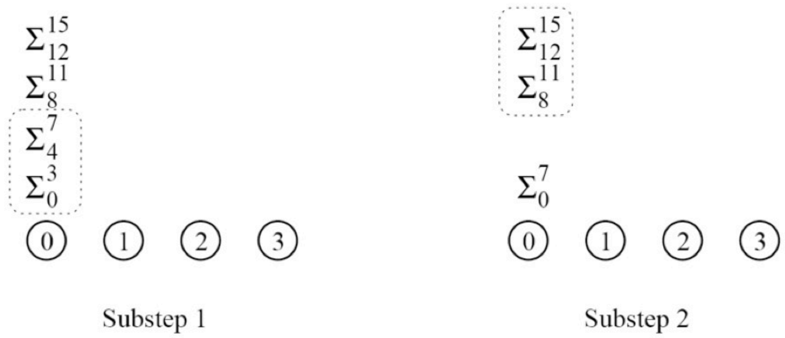
Improving granularity: the example of the sum, contd.



Improving granularity: the example of the sum, contd.



Improving granularity: the example of the sum, contd.



(c) Simulation of the third step in two substeps

(d) Simulation of the fourth step (e) Final result

Improving granularity: the example of the sum, contd.

- **Execution cost**
  - *log p* of the *log n* steps of the original algorithm
    - simulated in *(n / p) log p* steps on the *p* processors
  - The last log *n* - *log p* steps: no communications

- Total parallel execution time =

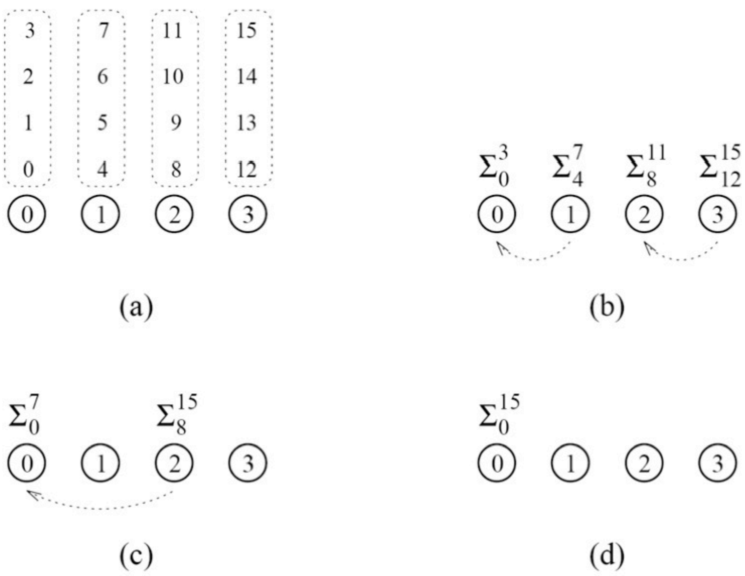
$$T_p = \Theta\ ((n / p) \log p)$$

- Total cost

$$p\ T_p = \Theta\ (n \log p)$$

  - Asymptotically > Θ *(n)* to add *n* numbers sequentially
  - The parallel system is therefore not cost optimal

Adding numbers in a cost-optimal fashion

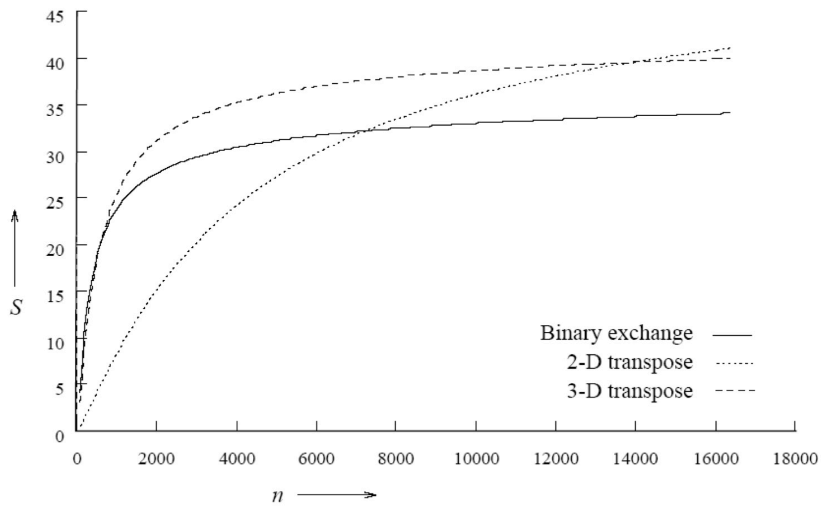


- Local sum: Θ *(n / p)*
- Partial sum: Θ *(log p)*
- Tp = Θ *(n / p + log p)*
- Cost = Θ *(n + p log p)*
- As long as *n* = Ω *(p log p)* , the cost is equal to Θ *(n)*

Scalability of parallel systems

- **Extrapolate performance**
  - How to move from a small problem on a small system
  - to a big problem on a larger configuration
- **Examples:** 3 algorithms to compute a n-point FFT on 64 processors

- Choosing this algorithm depending of configurations



Scalable parallel systems

- Total overhead function $T_o(T_s, p)$
  - Best sequential time T_s
  - Number of processors p

$$T_o = pT_p - T_s$$

- Efficiency

$$E = T_s / pT_p = T_s / (T_o + T_s) = 1 / (1 + T_o / T_s)$$

- Often, we have $T_o(T_s, p) / T_s < 1$
  - T_o grows in a sub-linear manner with respect to T_s
  - In this case, the efficiency increases if the size of the problem increases and if the number of processors is constant
- For such systems, it is possible to keep a constant efficiency by
  - Increasing the size of the problem
  - Increasing the number of processors proportionally
- Such systems are **scalable**

Scalability of parallel programs

- In scientific papers we read observations such as

  "We implemented an algorithm on the parallel machine X which obtained an acceleration of 10.8 out of 12 processors with a problem size equal to 100."

- A dot on a curve!
  - What happens if we have 100, 1000 processors?
  - What happens if we have data of size 10, 1000?

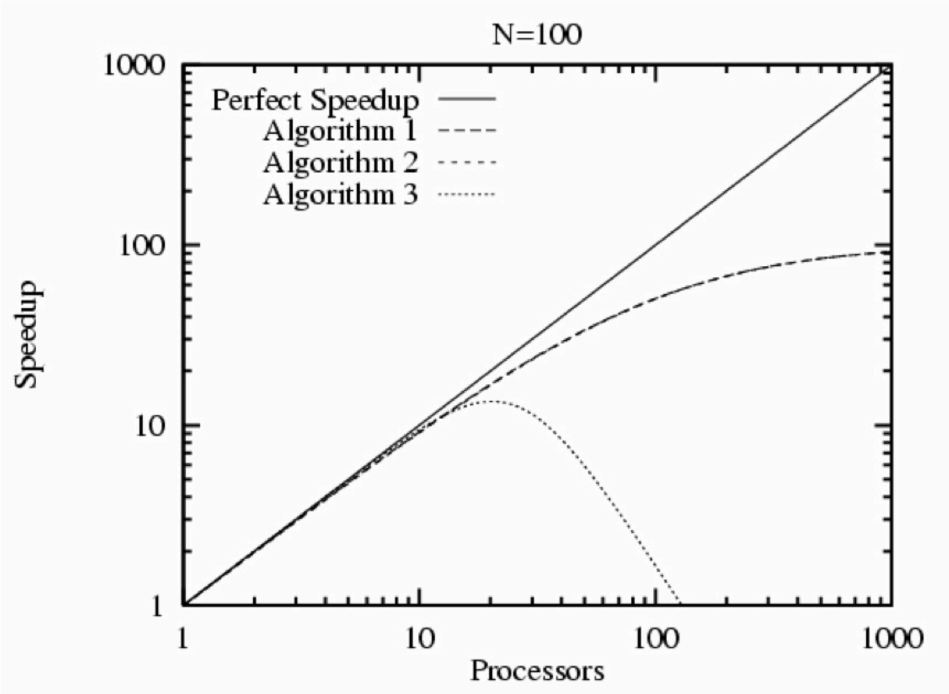
Scalability of parallel programs, contd.

- Three theoretical performance models
  - $T = N + N^2 / P$
    - This algorithm splits N^2 computations but also replicates N other computations
    - No other sources of additional cost
  - $T = (N + N^{2)} / P + 100$
    - This algorithm splits all the computations and adds an additional cost of 100
  - $T = (N + N^{2)} / P + 0.6 P^2$
    - This algorithm splits all the computations and adds an additional cost of $0.6 P^2$

- All these algorithms have an acceleration of 10.8 on 12 processors for $N = 100$!

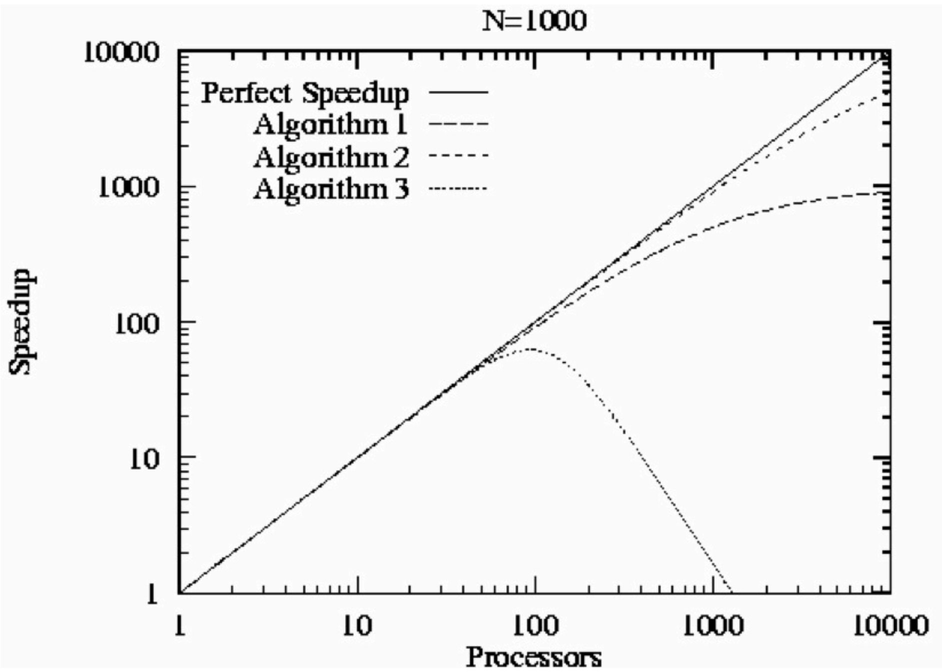
Scalability of parallel programs, contd.

If we increase the number of processors for N = 100



Scalability of parallel programs, contd.

If we increase the number of processors for N = 1000



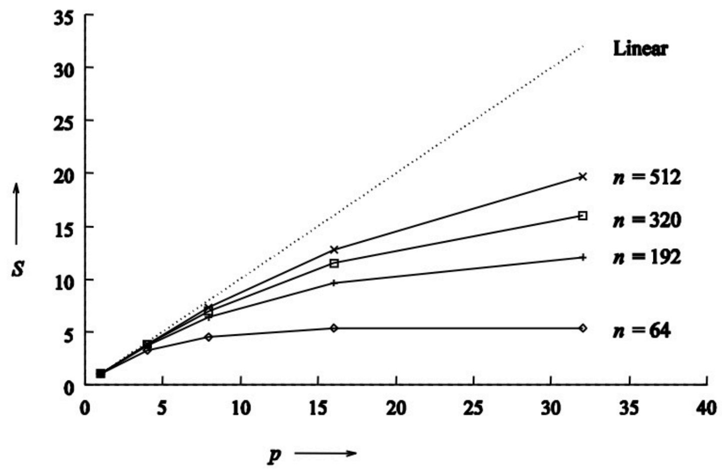
Scalability of parallel programs, contd.

- Adding *n* numbers on *p* processors
- Supposition: addition = communication = 1 time unit

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

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

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

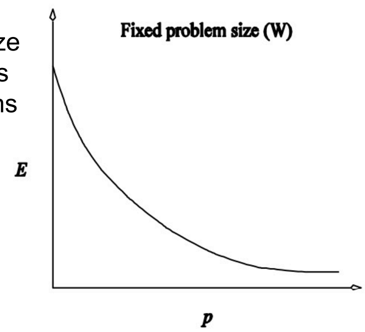


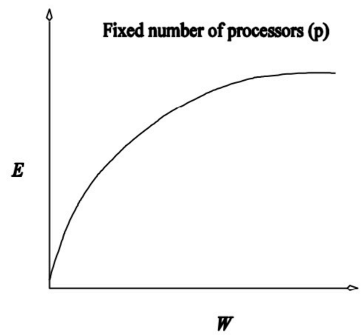
Acceleration tends to saturate and efficiency decreases

Scalability of parallel programs, contd.

- **Scalability and cost optimality are related**
  - In cost-optimal parallel systems, the efficiency = *Θ (1)*
  - Any parallel system can be made cost optimal
  - **Great care** must be taken to choose
    - The size of the computations
    - The number of processors used
- **Scalability and efficiency**

- Fixed problem size
- # Procs increases
- All parallel systems





- Problem size's increases
- Fixed number of processors
- Some parallel systems

Isoefficiency

- Rate at which the size of the problem must grow for each processor added to maintain a fixed efficiency

- Determines system scalability
  - The lower the rate the better

- To formalize this rate, define
  - Problem size W = asymptotic number of operations of the best sequential algorithm to solve the problem

Scalability metrics as a function of W

Parallel execution time

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

Speedup

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

Efficiency

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

Isoefficiency for scalable parallel systems

- To maintain efficiency at a set value (between 0 and 1)
- Maintain the rate $T_o(W, p) / W$ with a constant value
- For a desired efficiency value E

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

- Let $K = E / (1 - E)$ be a constant connected to a desired efficiency
- As T_0 is a function of W and p, we have

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

Isoefficiency function

- Rate of increase of W with respect to p to maintain a constant efficiency
  - Function of problem size W as a function of p
  - Can be obtained by algebraic manipulation

- Determines the ease with which a parallel system
  - Can maintain constant efficiency
  - Can attain increasing speedups in relation to the number of processors increasing

Asymptotic isoefficiency: example 1

Adding *n* numbers on *p* processors

- The overhead T_0 is approximately equal to $2p \log p$
  - $\log p$ levels in the tree
  - Communication and addition at each step

- For isoefficiency, we want to have $W = K\, T_0(W, p)$
- By replacing T_0 by $2p \log p$, we obtain $W = K\, 2p \log p$
- This gives an asymptotic isoeficiency equal to $\Theta(p \log p)$

- We want to obtain the same efficiency on p' processors as on p
  - When the number of processors changes from p to p'
  - The size of the problem n must increase by $(p' \log p')(p \log p)$

Asymptotic isoefficiency: example 2

Gaussian elimination for a matrix of size *n* on *p* processors

- Execution time = $O(n^3/p + n^2 + n \log p)$
- Total parallel cost = $O(n^3 + pn^2 + pn \log p)$
- Overhead $T_o = O(pn^2 + pn \log p)$
  - Pivot computation + backward substitution
- For the isoefficiency, we want to get $W = K\, T_o(W, p)$
- Expressing the overhead as a function of $W = n^3$ gives

$$T_o = O(pW^{2/3} + pW^{1/3} \log p)$$

- Asymptotic isoefficiency $W = K(pW^{2/3} + pW^{1/3} \log p)$
- We want to have the same efficiency with p' procs than with p procs
  - Using the 1st term $W = KpW^{2/3} \rightarrow W = K^3 p^3$
  - Using the second term $W = KpW^{1/3} \log p \rightarrow W = K^{3/2} (p \log p)^{3/2}$
  - The first term dominates: the work must increase with $(p')^3 / p^3$
    - The problem size should increase with p' / p

Asymptotic isoefficiency: example 3

A more complex example:

$$T_o = p^{3/2} + p^{3/4} W^{3/4}$$

- By using the 1st term of T_o:

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

- By using the 2nd term of T_o:

$$W = K p^{3/2} W^{3/4}$$
$$W^{1/4} = K p^{3/4}$$
$$W = K^4 p^3$$

- The largest of these asymptotic rates determines the isoefficiency
- The asymptotic isofficiency is therefore equal to $\Theta(p^3)$

Cost optimality and isoefficiency

- A parallel system is cost optimal iff

$$pT_p = \Theta(W)$$

- Thus we have

$$W + T_o(W, p) = \Theta(W)$$
$$T_o(W, p) = O(W)$$
$$W = \Omega(T_o(W, p))$$

- If we have an isoefficiency function $f(p)$
  - $W = \Omega(T_o(W, p))$ must be satisfied to ensure a cost optimality of a parallel system when its size increases

Lower bound for isoefficiency

- For a problem consisting of *W* work units
  - No more than *W* processors can be used optimally in cost
- To maintain a fixed efficiency
  - The problem size can increase at the speed of $\Theta(p)$
  - $\Omega(p)$ is the asymptotic lower bound of the isoefficiency function

Introduction to parallel Computing, 2nd Edition, A. Grama, A. Gupta, G. Karypis, V. Kumar, Addison Wesley

Minimum execution time

- Find the minimum time (T_P^{min}) to find a solution to a problem with a computing volume W
  - Derive the expression of the parallel time (T_P) with respect to the number of processors and equal it to 0

$$\frac{\partial T_P}{\partial p} = 0$$

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

Minimum execution time: example

Minimum time for the n numbers sum

- Parallel execution time

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

- Computing the derivative with respect to *p*

$$\frac{\partial}{\partial p}\left(\frac{n}{p} + 2\log p\right) = -\frac{n}{p^2} + 2\left(\frac{1}{p}\right)$$

- Equating the derivative to 0, solve for *p*

$$-\frac{n}{p^2} + 2\left(\frac{1}{p}\right) = 0$$

$$-\frac{n}{p} + 2 = 0$$

$$p = \frac{n}{2}$$

- The corresponding time is equal to

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