A Short Course in Parallel Scientific Computing

Technical University of Budapest

Graham Horton, University of Erlangen-Nürnberg

Talk 2 : Fundamentals of Parallel Computing

Contents

- Approaches to parallelism
- Measures for parallel algorithms
- Considerations for possible speedups
- Communication

Divide and Conquer (1)

Divide and Conquer is an important principle of many algorithms

The problem is partitioned into smaller subproblems

Solution is an appropriate combination of the subproblem solutions

The strategy is continued recursively until the subproblems become trivial

Example : Summing the elements of a vector

$$y = \sum_{i=1}^{n} x_i$$

Usual method:

$$\boxed{\begin{array}{l} y = 0 \\ \texttt{for } i \texttt{ = 1 to } n \texttt{ do} \\ \qquad y = y + x_i \end{array}}$$

Recursion :

$$\sum_{1}^{n} x_i = \sum_{1}^{\frac{n}{2}} x_i + \sum_{\frac{n}{2}+1}^{n} x_i$$

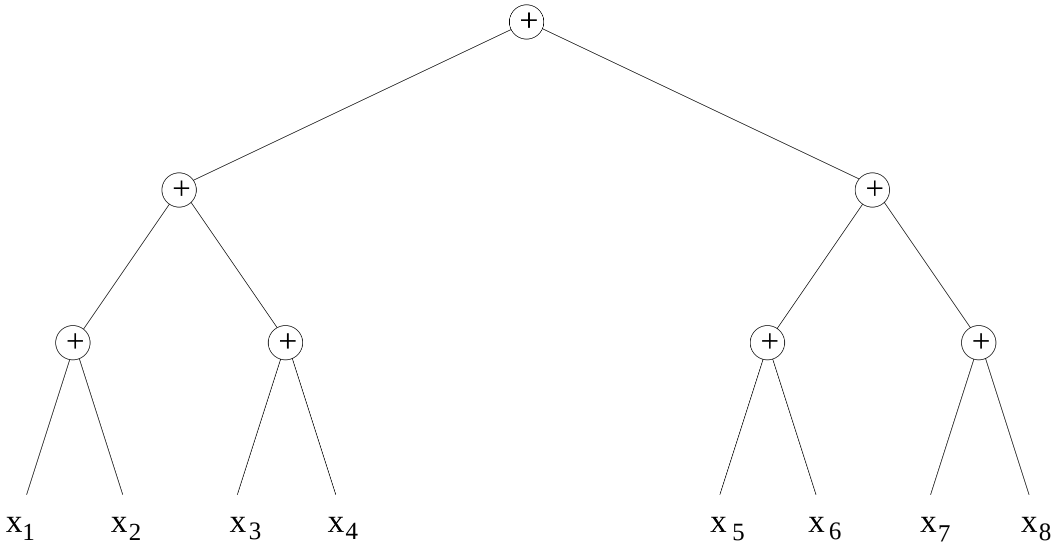
Analogously maximum computation:

$$\max_{1 \le i \le n} x_i = \max(\max_{1 \le i \le \frac{n}{2}} x_i, \max_{\frac{n}{2}+1 \le i \le n} x_i)$$

In general: every binary, associative operator

Divide and Conquer (2)

Represent the computation as a binary tree:



All subproblems of each level can be processed simultaneously

Sequential computation time : $n-1$ units

Parallel computation time : $\log_2 n$ units with $\frac{n}{2}$ processors

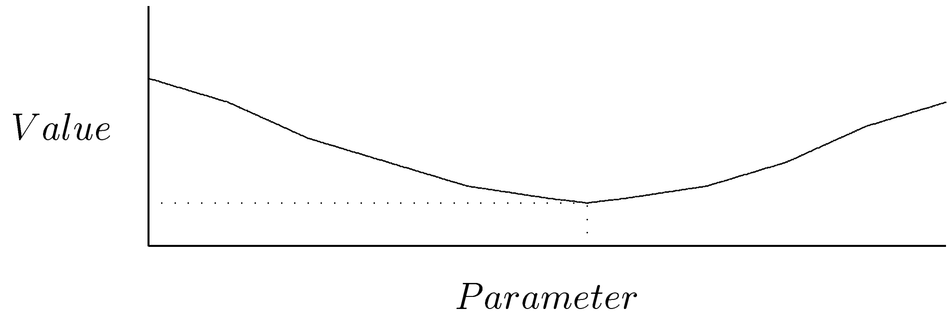
Such operations are very common

Problem : low degree of parallelism on most levels

Parameter Parallelism

Compute for what paramater value a certain event occurs

In particular optimisation problems



Examples :

- What angle of attack of an aircraft wing yields maximum lift?
- Which mix gives optimal fuel efficiency?

Solve a complex problem for each parameter value

Almost trivial parallelism, as the computations are independent

Each processor solves complete problem for one choice of parameters

No communication necessary during computation

Questions: Load balancing? Exceptions?

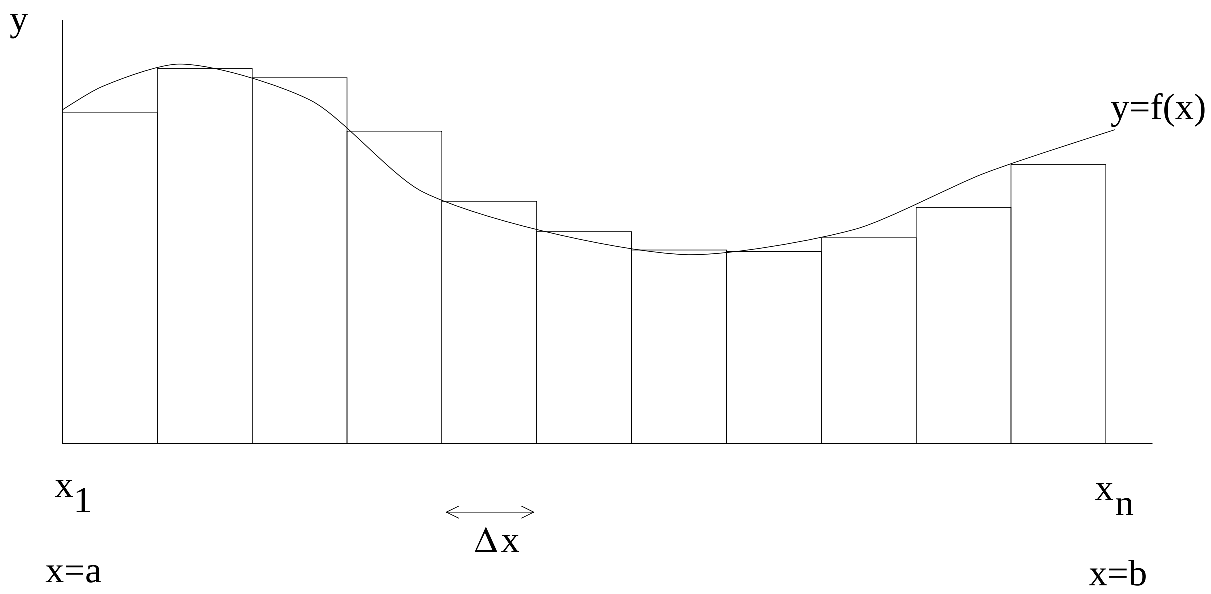
Data Parallelism

We must perform the same operation on many items of data

Example : Approximation of π by the integral

$$\pi = 4 \int_0^1 \frac{1}{x^2+1} \quad dx$$

In general: numerical integration of $y = f(x),\ x \in [a, b]$



Subdivide the x-axis into discrete points $x_1, \ldots, x_n$

Approximation of the curve by straight line segments

Integration by computing the area of the rectangles:

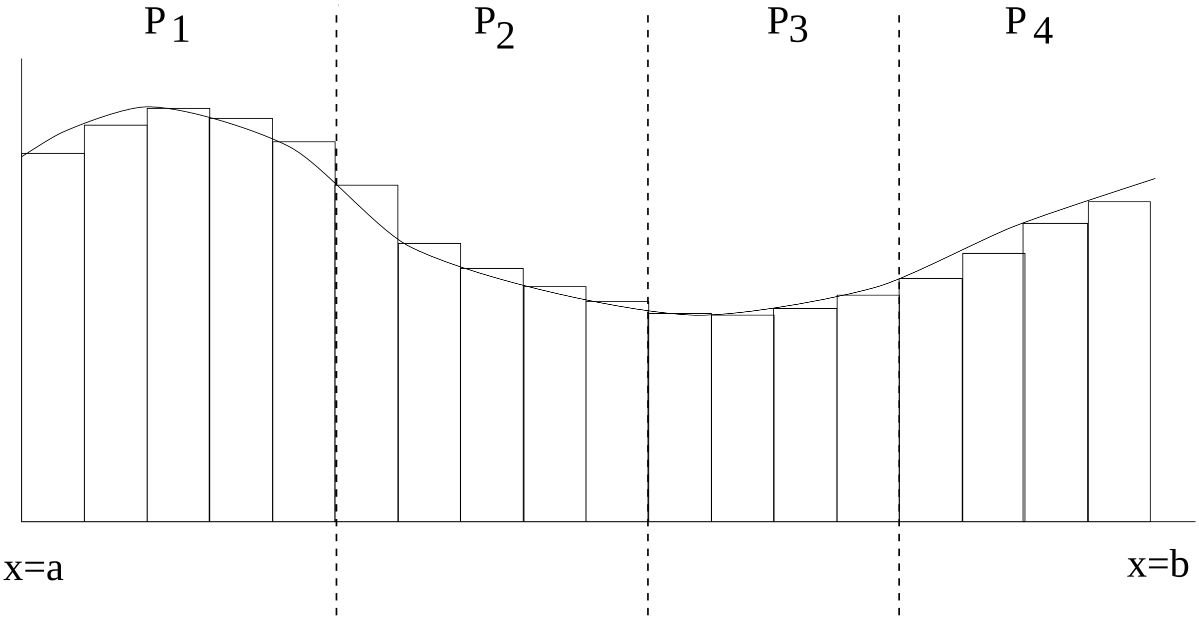
$$\int_{x=a}^{x=b} f(x) \quad dx \approx \Delta x \sum_{1}^{n-1} f(x_i)$$

Example of a Data-Parallel Computation

The sequential computation consists of

- $n-1$ evaluations of $f(x)$
- $n-2$ additions
- 1 multiplication

Partitioning the $x_1, \ldots, x_{n-1}$ onto p processors



The parallel computation now takes

- $\lceil \frac{n-1}{p} \rceil$ evaluations of $f(x)$
- $\lceil \frac{n-2}{p} \rceil + \lceil \log_2 (p) \rceil$ additions
- 1 multiplication

Levels of Parallelism

We distinguish parallelism at different levels

1. Sub-operation level (vector operations)
2. Instruction level (dataflow processor, superscalar processor)
3. Block level (PARALLEL DO, etc.)
4. Process level (multi-tasking)
5. Program level (computer centre)

The *granularity* of parallelism is usually

- *fine* in 1 and 2
- *medium* in 3
- *coarse* in 4 and 5

Parameter parallelism is in 4 or 5

Divide and Conquer is 3

Data parallelism is usually 1, 2, or 3

Measures for Parallel Algorithms

Measures serve to judge the effectiveness of a parallel program

There are many measures in use

Most important are *Speedup* S and *Efficiency* E

If $T(p)$ is the computation time for a program on p processors, then

$$S(p) = \frac{T(1)}{T(p)}, \qquad E(p) = \frac{S(p)}{p} = \frac{T(1)}{pT(p)}$$

Speedup is therefore the acceleration factor of the program

Efficiency measures the effective utilisation of the processors

We have

$$0 \leq S(p) \leq p, \qquad 0 \leq E(p) \leq 1$$

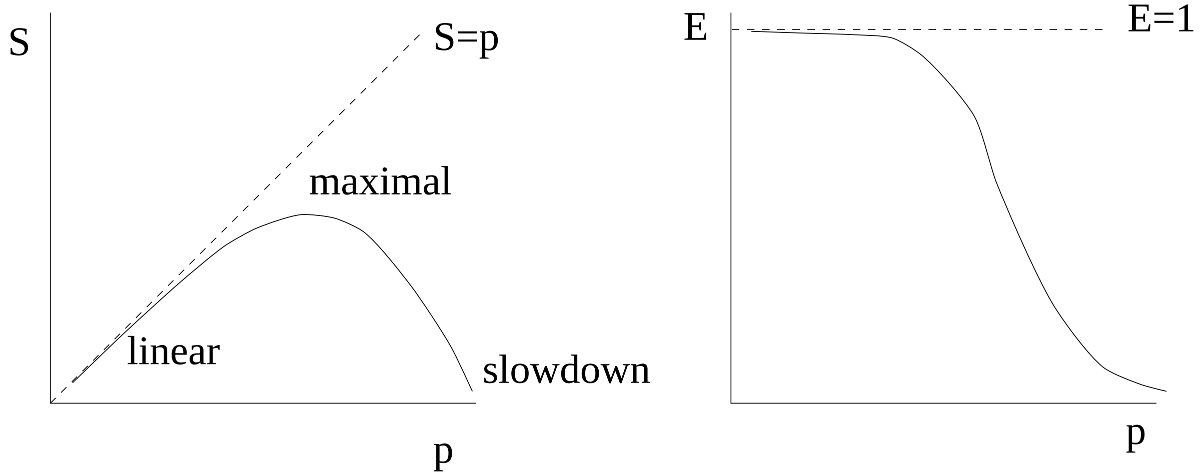
Can $S > p$ be true?

Possible "counterexamples":

- Parallel search
- Loss of loop and index overheads
- Finding the fastest of n programs

Measures for Parallel Algorithms

Speedup and Efficiency typically show following behaviour



In the linear portion

- Communication costs are low
- Speedup is almost optimal

Beyond the maximal speedup delays dominate

- Communication / Synchronisation
- Unequal load balance

Speedup < 1 ("slowdown") is also possible!

Scaled Speedup

It has been said that Speedup and Efficiency are irrelevant

Solving the *same* problem with more processors is unrealistic

Rather, one would use more processors to solve a larger problem

Assumption: we want to solve larger problems than is currently possible

Therefore: scale the problem with p

Example : Weather forecasting

This gives the *Scaled Speedup* :

$$S_S(p) = \frac{pT(1)}{T(p)}$$

The *Scaled Efficiency* is defined analogously :

$$E_S(p) = \frac{T(1)}{T(p)}$$

N.B. $T(p)$ refers to a p-times larger problem!

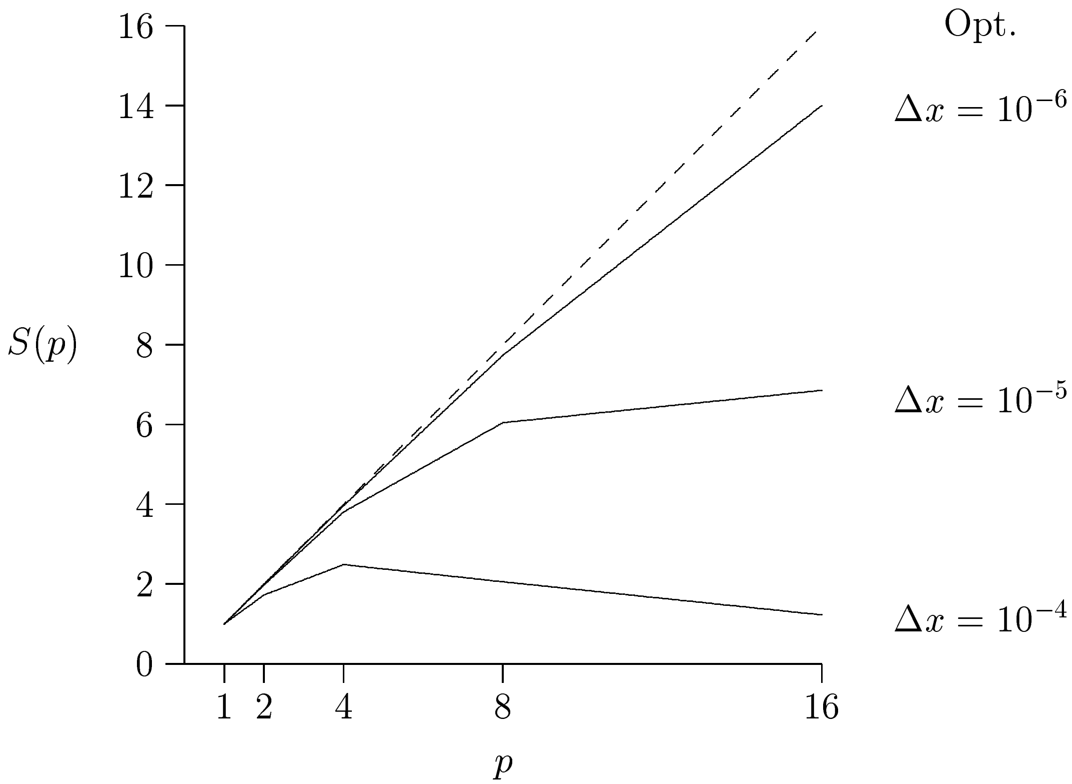
Is supposed to give a truer picture of parallelisability of an algorithm

J. L. Gustafson : *Reevaluating Amdahl's Law.*
Communications of the ACM, Vol. 31, No.5, May 1988, pp.532-533

Examples of Speedup and Efficiency

Consider the example "parallel approximation of π"

Experimental results (Speedup) :

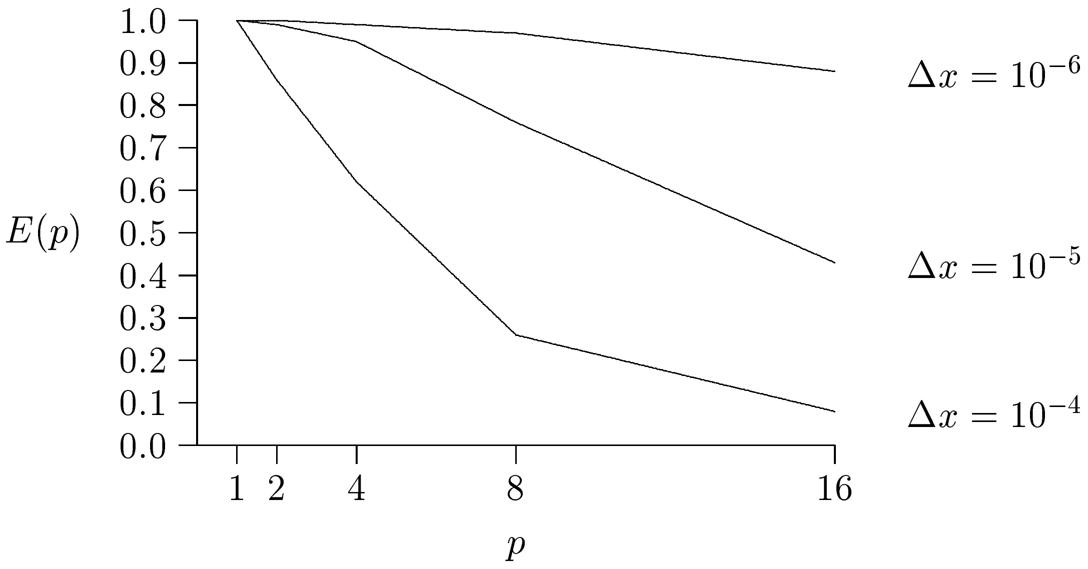


Linear behaviour for large problem and small p

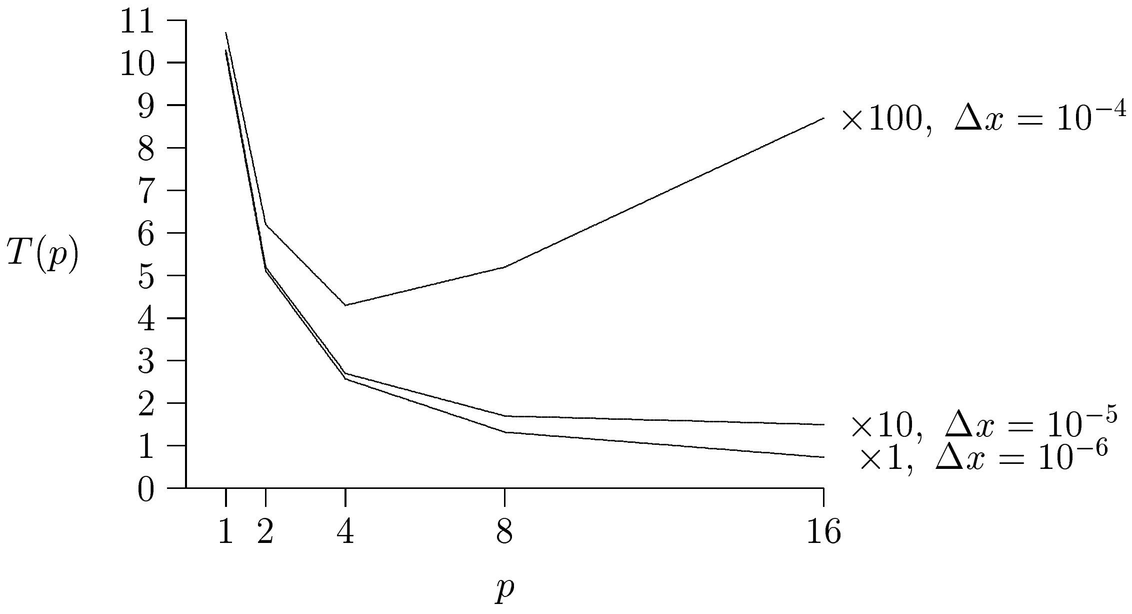
Performance deteriorates for larger p and smaller problems

Examples of Speedup and Efficiency

Efficiencies :



Computation times:



Computation times are normalised

N.B. Increase in time with p for smallest problem!

This is the first programming exercise

Where do we lose Efficiency?

Efficiency $< 100\%$ can arise for the following reasons

- Computational overhead
  - Introduction of extra operations in order to achieve parallelism
  - Example : Solution of a tridiagonal system
- Load imbalance
  - Load imbalance leads to idle times
- Communication
  - Processors must exchange data
  - During this time, no computation can take place (?)
  - Unavoidable, must be minimised
- Sequential fraction
  - Some parts of an algorithm cannot be parallelised
  - Example : Input/Output

Amdahl's Law (1)

Amdahl made the following observation in 1967:

Let T_s be the *sequential fraction* of an algorithm

$$T(1) = T_s + (1 - T_s) \quad (= T_s + T_p)$$

Then the parallel computation time must obey

$$T(p) = T_s + \frac{1 - T_s}{p}$$

From which follows

$$S(p) = \frac{1}{T_s + \frac{1 - T_s}{p}} \leq \frac{1}{T_s}$$

i.e. the maximal speedup is bounded by the sequential fraction

Example : Matrix multiplication

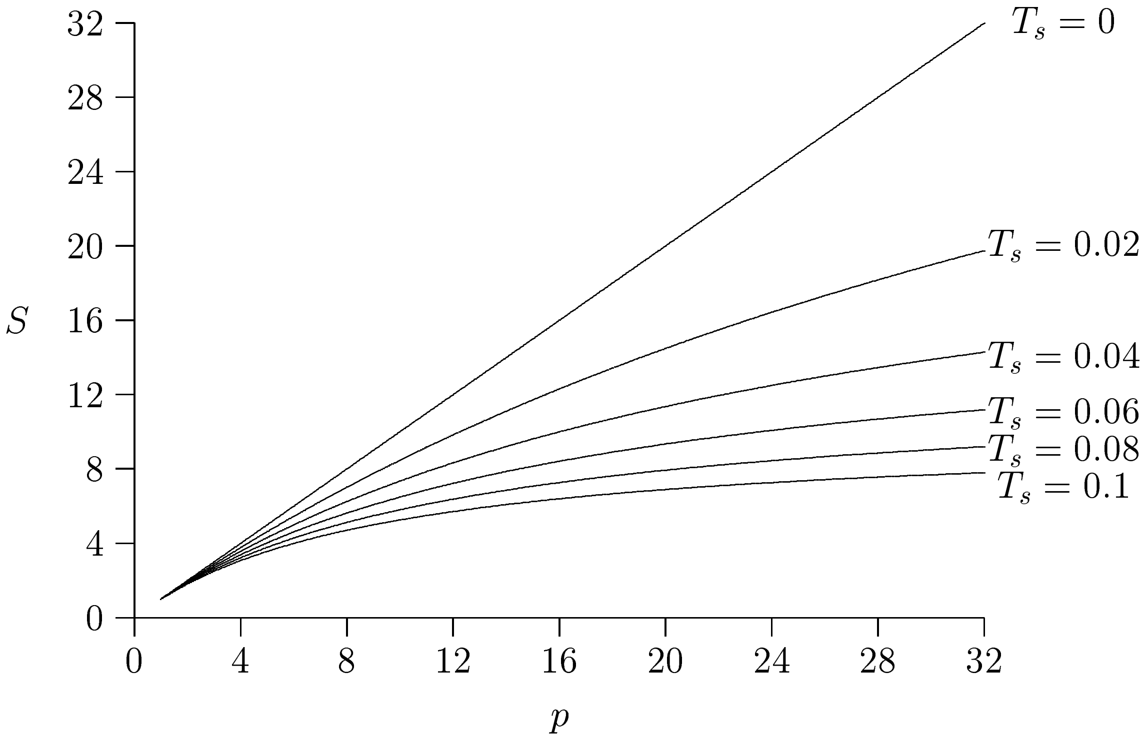
$T_s =$ Reading in the data

$T_p =$ Performing the arithmetic operations

Amdahl's Gesetz (2)

To achieve high efficiency, algorithm must have very small T_s

Values of $S(p)$ for different T_s :



The argument is valid, but the consequences are small in practice

Reasons:

- T_s is very often extremely small
- Argument does not hold for scaled speedup

G. M. Amdahl : *Validity of the single processor approach to achieving large scale computing capabilities.*
AFIPS Proc. of the SJCC, Vol. 31, 1967, pp.483-485

Minsky

Minsky made the following famous statement in 1970:

> In multiprocessor systems the Speedup will grow
> not with p, but only with $\log(p)$, apart from
> in a few cases.

This is too pessimistic!

The "few cases" are the most compute-intensive ones!

Many problems especially in Scientific Computing belong there

The result follows from Amdahl's Law, if we set

$$T_s = \frac{c}{\log p}$$

for some constant c

M. Minsky : *Form and content in computer science.*
Journal of the ACM, Vol. 17, No.2, April 1970, pp.197-215

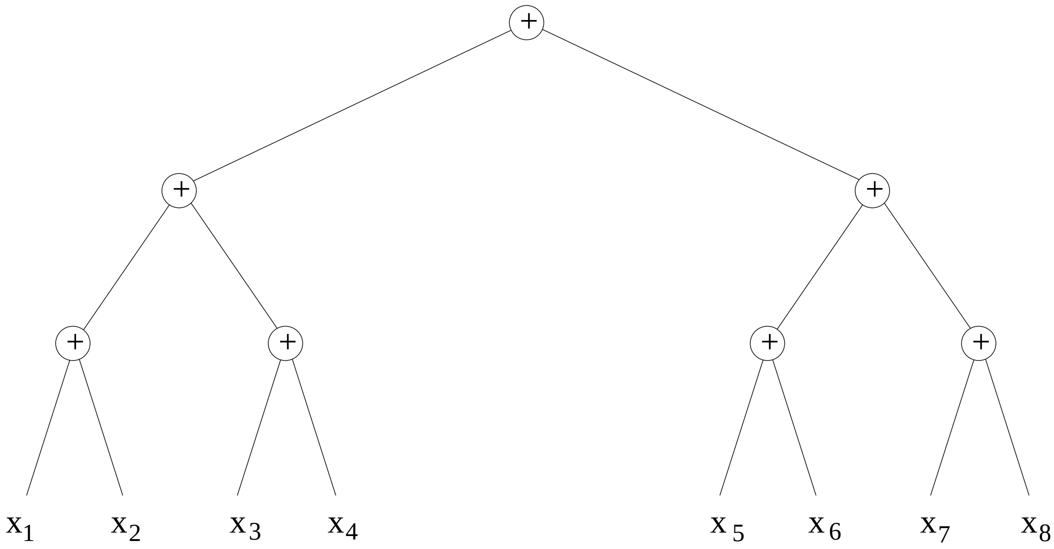
Lee

Lee stated the following in 1977:

$$S(p) \leq \frac{p}{\log p}$$

This is valid for many parallel algoritms

Example: Addition of $n = 2^m$ numbers with $p = \frac{n}{2}$:



Sequential operations : $n - 1$

Parallel operations : $\log_2(n)$

$$S(p) = \frac{n-1}{\log_2(n)} = O\left(\frac{p}{\log(p)}\right)$$

Communication

Every parallel algorithm needs to transfer data between processors

In most (MIMD) multiprocessors this requires sending a message

Formula for the time T to send a message

$$T = \alpha + \beta n$$

mit

- n, the message length in bytes
- α, the latency time
- β, the transfer time per byte

This is true for neighbouring processors

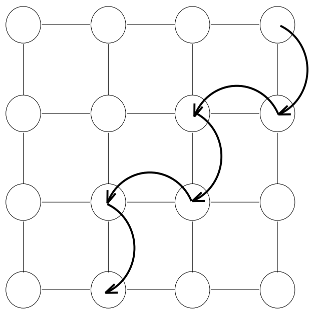
Consequences:

- Large value of α means many messages become expensive
- Large value of β means that large messages are expensive
- It is better to combine several small messages into one larger one

Communication

Often the multiprocessor will have local neighborhood topology

Example : Meiko Transputer system



Long-distance messages pass though intermediate processors

For non-neighbouring processors we therefore have

$$T = h(\alpha + \beta n)$$

where h is the number of "hops" between communicating processors

Consequences:

- as previous page, plus
- Map communicating subproblems to neighbouring processors