Performance Evaluation, etc

(cf. Grama et al.)

Amdahl’s Law

- **Black beast** of researchers in the field of parallel computing until 1988
- It “**limited”** the speedup of a parallel machine to 20-50, regardless of the number of processors!
- In 1988, however, some researchers from Sandia National Labs obtained speedup of 1000 on a 1024 processors machine **without violating** the Amdahl's Law. How did they do?

Speedup

Measure of how much faster the computation executes versus the best serial code

- Serial time divided by parallel time

Example: *Painting a picket fence*

- 30 minutes of preparation (serial)
- One minute to paint a single picket
- 30 minutes of cleanup (serial)

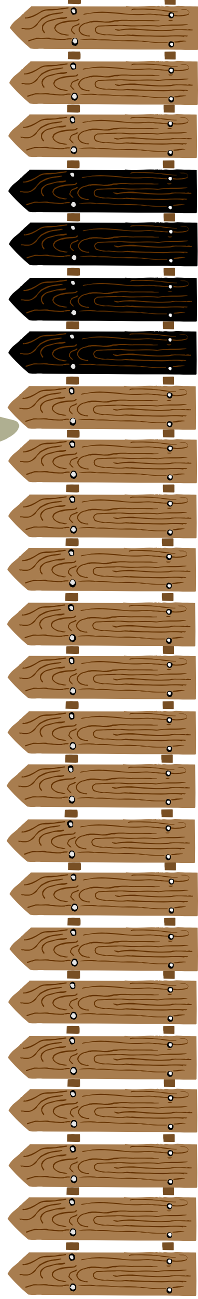
Thus, 300 pickets takes 360 minutes (serial time)

Computing Speedup

Number of painters	Time	Speedup
1	30 + 300 + 30 = 360	1.0X
2	30 + 150 + 30 = 210	1.7X
10	30 + 30 + 30 = 90	4.0X
100	30 + 3 + 30 = 63	5.7X
Infinite	30 + 0 + 30 = 60	6.0X

Illustrates Amdahl's Law

Potential speedup is restricted by serial portion

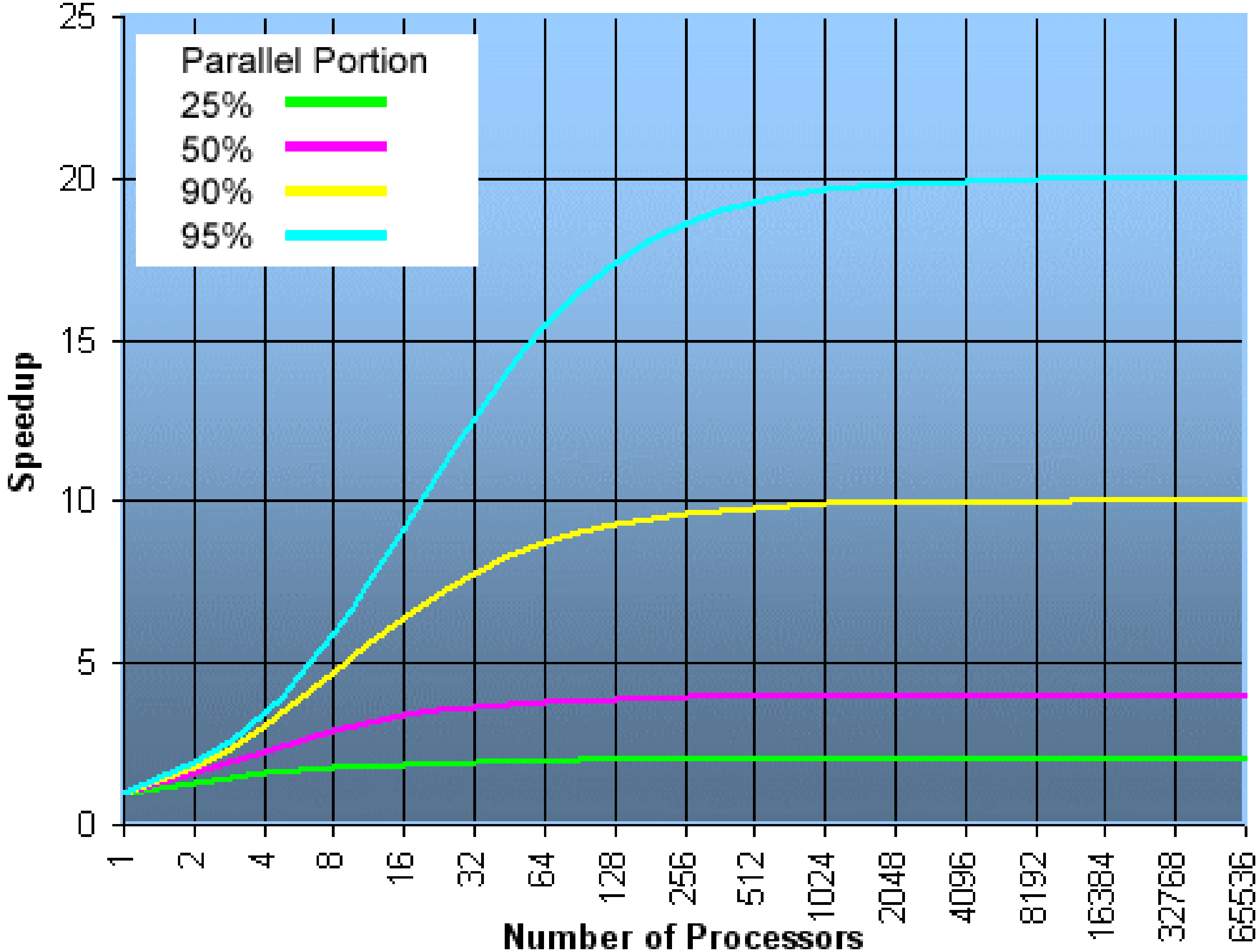


What if fence owner uses spray gun to paint 300 pickets in one hour?

- Better serial algorithm
- If no spray guns are available for multiple workers, what is maximum parallel speedup?

Amdhal's law

$$Speedup = \frac{1}{\frac{F_p}{N} + F_s}$$



Scalable problem

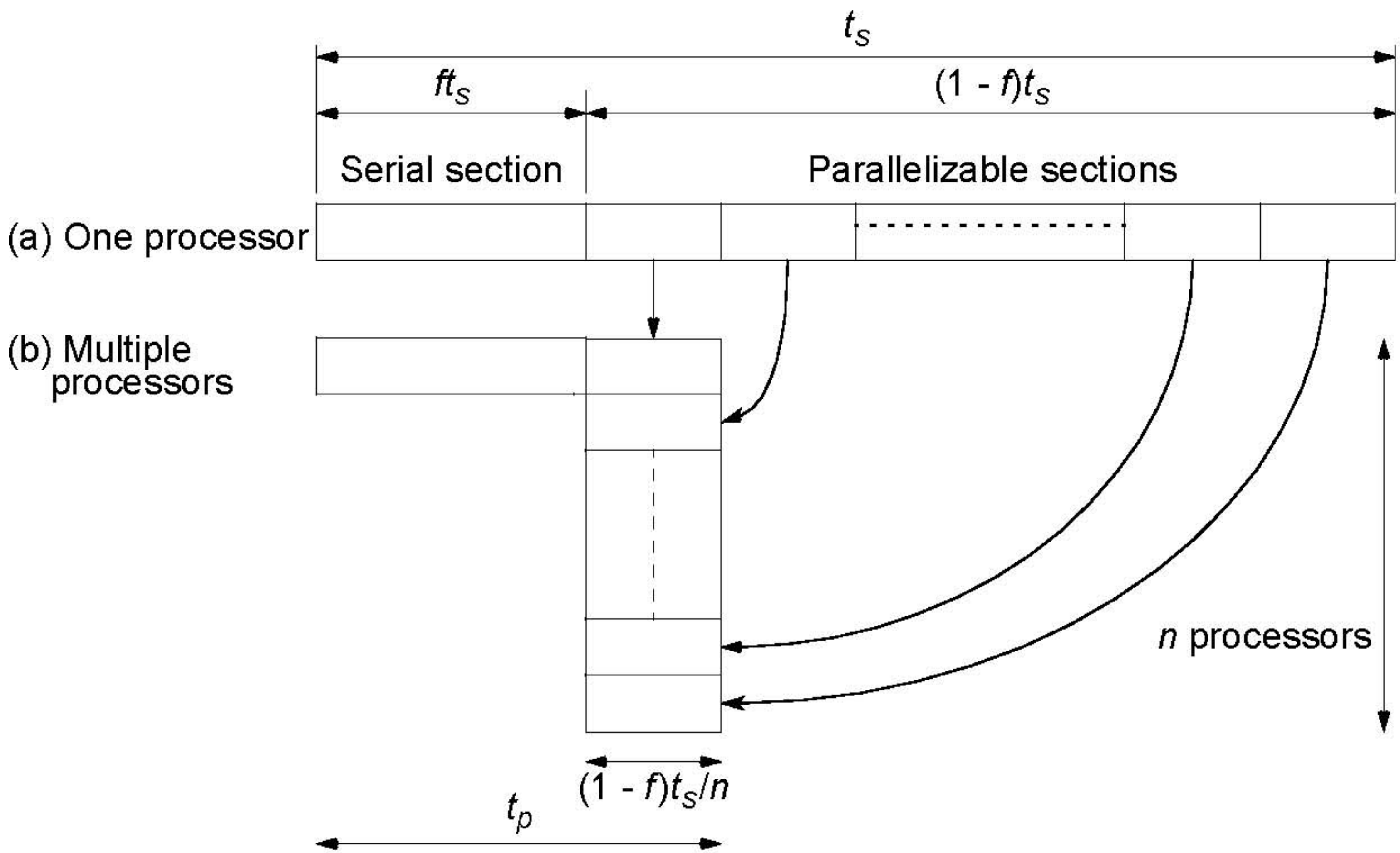
- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time
- Example:
  - certain problems demonstrate increased performance by increasing the problem size

2D Grid Calculations	85 sec	85%
Serial fraction	15 sec	15%

- Increase the problem size by doubling the grid dimensions and halving the time step.
  - four times the number of grid points
  - twice the number of time steps

2D Grid Calculations	680 sec	97.84%
Serial fraction	15 sec	2.16%

Maximum Speedup – Amdahl's law



Maximum Speedup – Amdahl's law

Speedup factor is given by:

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1 + (n-1)f}$$

This equation is known as *Amdahl's law*

Amdahl's Law

- Thus, for n $\rightarrow \infty$:

$$S(n) \rightarrow \frac{1}{f}$$

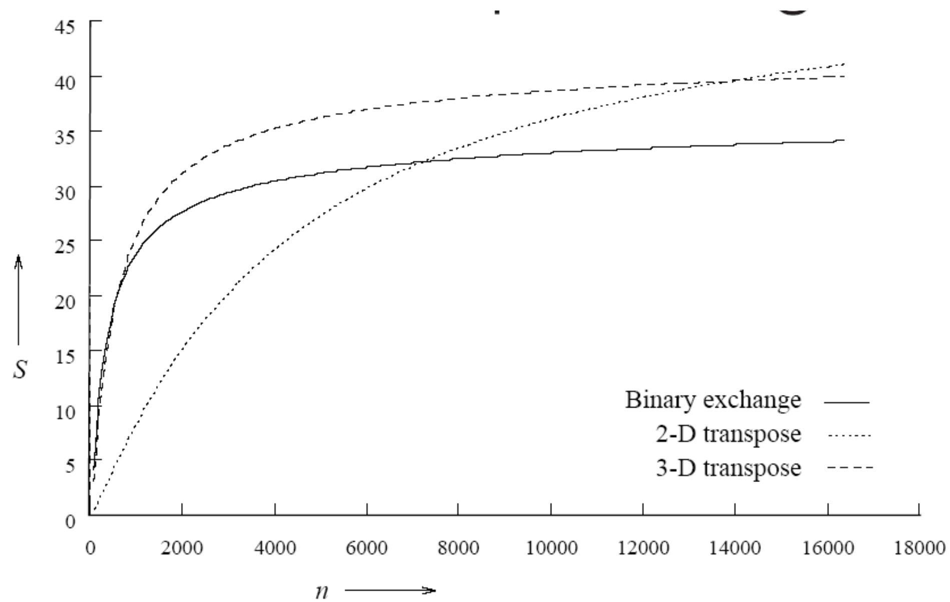
- For instance, if the serial fraction is f = 5% (very plausible!) the maximum speed-up is 20!

- OBS The **serial fraction** represents that "fraction" of code that can not be **parallelized** (eg I / O, critical sections, etc.)

Scalability

How do we **extrapolate the performance** from **small** problems and small systems to **larger** problems with **larger** configurations?

The following example shows the evolution of the speed-up of **three algorithms** for a n-point Fast Fourier Transform (FFT) on 64 processors



For small values of ***n***, it would seem that the Binary-exchange and 3-D transpose algorithms are the best, but for ***n* > 18000**, the 2-D transpose algorithm allows better speedup.

Moral: It is difficult to infer the scalability of data from observations and "small" machines

Scalability

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

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

or, by remembering the concept of parallel overhead $T_o = p\ T_P - T_S$, we have:

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

- For a given problem of dimension (i.e., the value T_S remains constant), if we increase the number of processors p , T_o **increases**

- At the contrary, the total **efficiency** of the program **decreases**. This is the case of all parallel programs.

Scalability : Example

- Consider the usual problem of adding **n** numbers on **p** processors (**intelligent scaling** algorithm)

- We have seen that (let's avoid asymptotic analysis, we work with constants):

$$T_P = \frac{n}{p} + 2\log p \longleftarrow \left\{ \begin{array}{l} \log p \text{ communication steps} \\ \qquad\qquad + \\ \log p \text{ addition steps} \end{array} \right.$$

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

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

Scalability : Example

If we visualize the speedup for various sizes of input, we get:

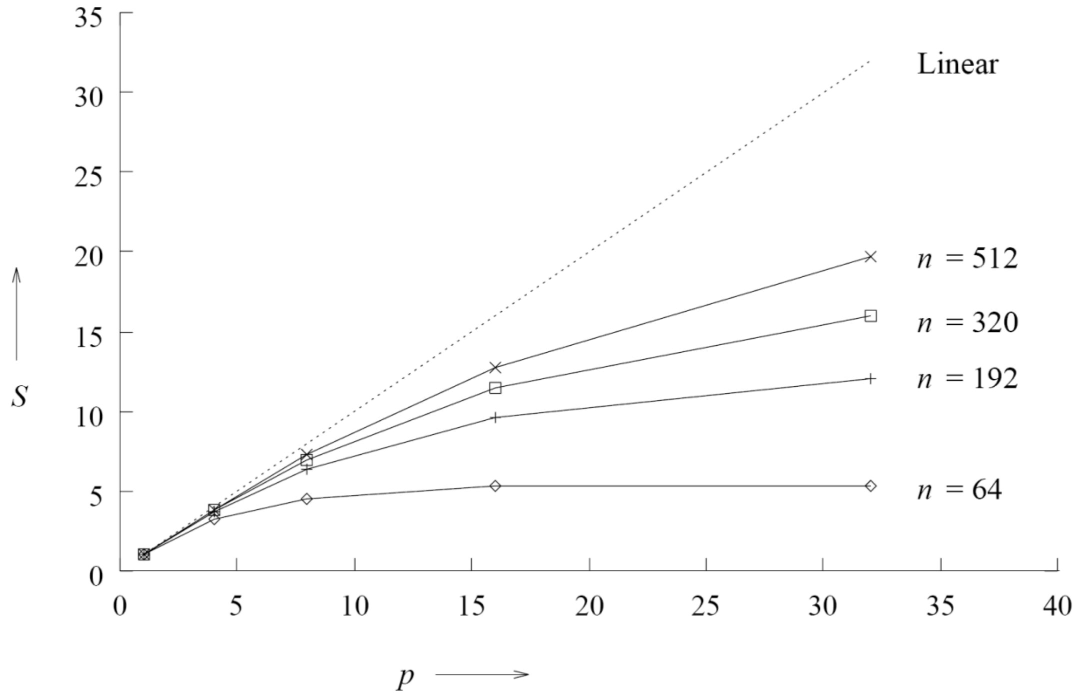


Figure 5.8 Speedup versus the number of processing elements for adding a list of numbers.

The speedup tends to **saturate** and the **efficiency decreases** as a consequence of **Amdahl's Law**

Scalability: Features

- The total overhead function ***To*** is a function of both ***Ts*** (ie, the size of the problem) and the number of processors ***p***
- In many cases, ***To*** grows sub-linearly with respect to ***Ts***
- In such cases, the **efficiency increases** if the dimensions of the problem are increased, keeping the number of processors constant
- For these systems, we can **simultaneously increase** the size of the problem and the number of processors to maintain constant the efficiency (but how?)
- Such systems are called **scalable parallel systems**

Scalability: Features

- In a scalable system, the efficiency remains constant with the increase of both ***p*** and ***n***.
- For example, in the case of the sum of ***n*** numbers on ***p*** processors, the efficiency remains constant at 0.8 if

$$\boldsymbol{n = 8\, p \log p}$$

Table 5.1 Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

NB: Starting from $\mathbf{0.8 = 1/(1+2}\boldsymbol{p\log p/n}\mathbf{)}$ the above relation is found …

Scalability: Conclusions

- Recall that parallel systems that are **cost-optimal** have efficiency Θ (1).

- Therefore, **scalability** and **cost-optimality** are closely related

- A **scalable** parallel system can be made **cost-optimal** if the number of processors and the size of the problem are chosen in an appropriate way

In summary:

- For a given size of the problem, if we **increase** the number of processors, the overall efficiency of the system **decreases** (for all systems)

- For some systems, **the efficiency of a parallel system increases if the dimensions of the problem are increased while the number of processes remains constant**

Isoefficiency

The two previous concept can be illustrated by the following chart

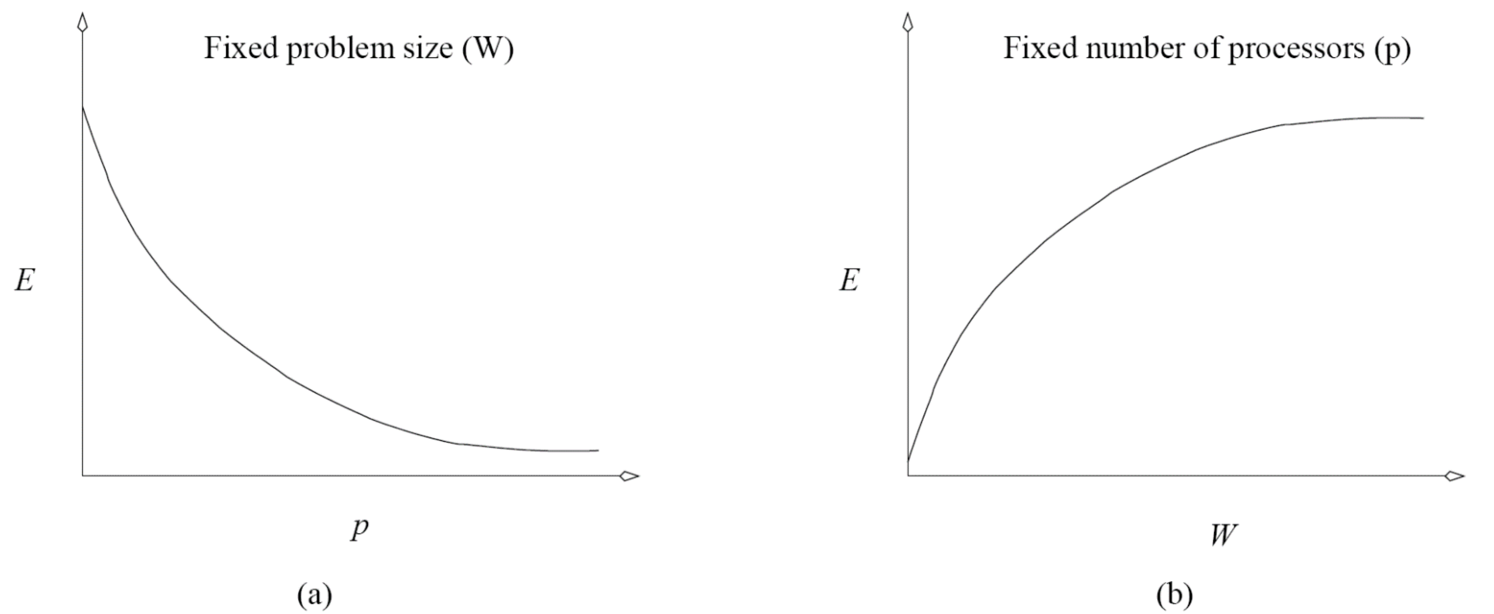


Figure 5.9 Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

Isoefficiency

- Which is the **growth rate** of the size of the problem with respect to the number of processors in order to keep the efficiency constant?

- This rate determines the **scalability** of the system. The smaller, the better

- Before formalizing this concept, we must formally define what are "**the size of the problem**"

Isoefficiency

- Possible solutions: **Size of input? Not good!**

- ES: Let ***n*** be the input size of a matrix computation problem (eg, sum of matrices, matrix-matrix product). If ***n*** is doubled, there will be an increase of 8 times for an operation involving the **matrix – matrix product**, while 4 times if the operation is a **sum of matrices**. So, it "depends" on the problem

- To this end, we define the **problem size W** the asymptotic **number of operations** associated with the best serial algorithm to solve the problem (in practice it is the **sequential time**!)

Isoefficiency: Formalism

- The parallel time can be written as (recall that $W = T_s$):

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

- Thus, speedup is

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

- Eventually, the expression becomes:

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

Isoefficiency: Formalism

- For scalable parallel systems, **efficiency** can be maintained at a fixed value (between 0 and 1), if the ratio ***To / W*** is kept constant
- For a given value and efficiency

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

- If ***K = E / (1 - E)*** is a constant that depends on the efficiency to be maintained, given that ***To*** is a function of ***W*** and ***p***, we have:

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

Isoefficiency: Formalism

- The previous formula takes the name of function **isoefficiency**

- From this formula, it is clear that the ***dimensions of the problem W*** can be obtained as a function of ***p*** through algebraic steps, in order to keep the **efficiency constant**

- This function determines the "**easiness**" with which a parallel system can maintain a constant efficiency and, consequently, achieve a **speedup** that increases in **proportion to the number of processors**

Isoefficiency: Example

- The **overhead function** for the problem of adding ***n*** numbers on ***p*** processors is approximated by **2p log p**

- By replacing ***To*** with **2p log p**, we obtain:

$$W = K2p\log p.$$

So, the asymptotic isoefficiency function for this parallel system is

.

$$\Theta(p\log p)$$

- If the number of processors is **increased** from p to p‘ (that is, by a factor of p' / p), the size of the problem (in this case, ***n***) must be increased by a factor **(p‘ log p') / (p log p)** in order to achieve the **same efficiency** of p processors.
- In this way, the speedup **increases** by a factor of **p'/ p**

Isoefficiency: Example (from the Pacheco book)

Consider the sequential version of the trapezoidal rule:

```
h = (b-a)/n;
integral = (f(a) + f(b))/2.0;
x = a;
for (i = 1; i <= n-1; i++) {
     x = x + h;
     integral = integral + f(x);
}
integral = integral*h;
```

The execution time can be approximated by

$$T_S(n) = c_1 + c_2\,(n\text{-}1) = k_1 n + k_2 \cong k_1 n$$

instructions "outside" the `for`

instructions “in" the `for`

n = trapezoid number

Isoefficiency: Example (from the Pacheco book)

We consider the parallel version of trapezoid rule:

```
h = (b-a)/n;      /* h is the same for all processes */
local_n = n/p;    /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);

/* Add up the integrals calculated by each process */
MPI_Reduce(&integral, &total, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);
```

The parallel execution time can be approximated by

$$T_p(n, p) = k_1 n/p + k_2\, log_2(p) + k_3$$

instructions "due" to the `Trap` call

Instructions “due” to `MPI_Reduce` (binary tree)

independent instructions from p and n (negligible)

Isoefficiency: Example (from the Pacheco book)

- Being

$$T_o = p\ T_P - T_S$$

- We have (neglecting k_3)

$$T_o = k_1 n + k_2\, p\ log_2(p) - k_1 n = k_2\, p\ log_2(p)$$

- From the definition of efficiency,

$$E = \frac{1}{1+\frac{T_O}{W}} = \frac{1}{1+\frac{k_2 p \log_2(p)}{k_1 n}} \Rightarrow n = \frac{E}{1-E}\frac{k_2}{k_1} p \log_2(p)$$

Isoefficiency: Example (from the Pacheco book)

- The formula $$E = \frac{1}{1+\frac{T_O}{W}} = \frac{1}{1+\frac{k_2 p \log_2(p)}{k_1 n}}$$

tells us that, in order to **maintain a constant efficiency**, "n should increase as $p \log_2(p)$"

- For example, both p = 4 and n = 512. If we want to maintain the same efficiency with 8 (i.e. 2 p) processors, we have to impose (<u>instead of doubling</u> n <u>as you might think</u>):

$$\frac{k_2 4 \log_2(4)}{k_1 512} = \frac{k_2 8 \log_2(8)}{k_1 n} \Rightarrow n = 1536$$