Performance Evaluation, etc

(cf. Grama et al.)

Amarcord

O(g(n))

Date due costanti positive c ed n_0, una funzione $f(n)$ appartiene all'insieme $O(g(n))$, ovvero $f(n) \in O(g(n))$ se:

$\exists\ c, n_0 > 0 \mid \forall\ n > n_0,\ 0 \leq f(n) \leq c\, g(n)$

Ω(g(n))

Date due costanti positive c ed n_0 , una funzione $f(n)$ appartiene all'insieme $\Omega(g(n))$, ovvero $f(n) \in \Omega(g(n))$ se:

$\exists\ c, n_0 > 0 \mid \forall\ n > n_0,\ f(n) \geq c\, g(n) \geq 0$

Θ(g(n))

Date tre costanti positive c_1, c_2 ed n_0 , una funzione $f(n)$ appartiene all'insieme $\Theta(g(n))$, ovvero $f(n) \in \Theta(g(n))$ se:

$\exists\ c_1, c_2, n_0 > 0 \mid \forall\ n > n_0, 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$

Amarcord

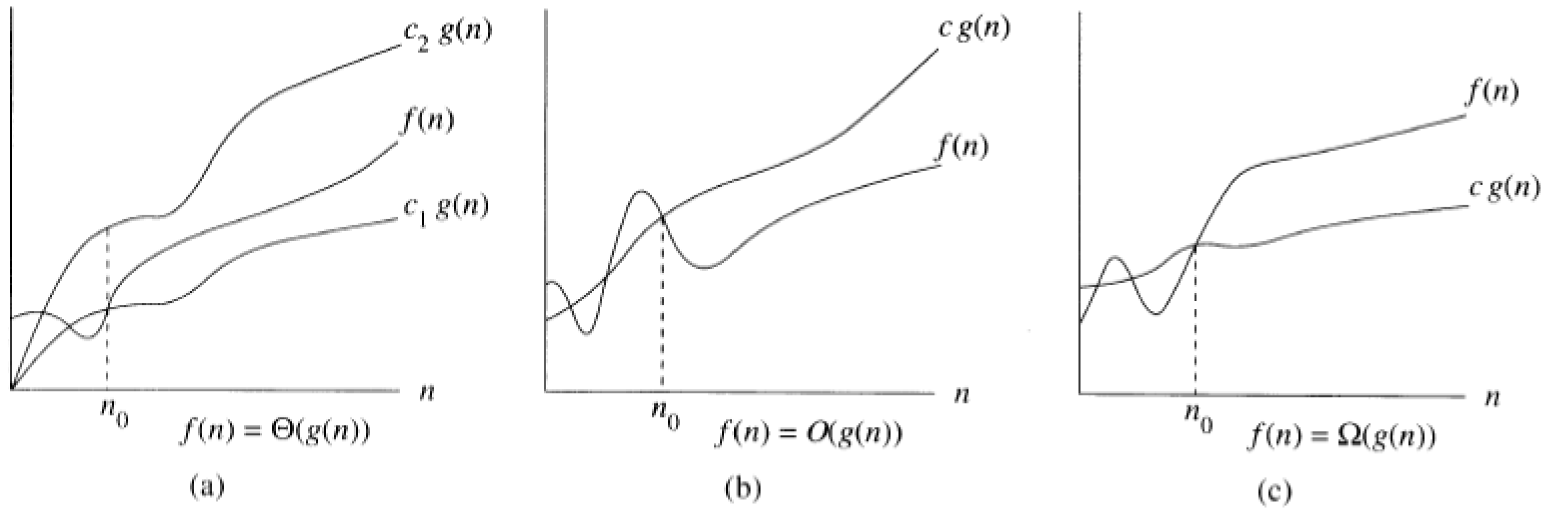


Fig. 1 : Esempi di funzioni $f(n)$ che appartengono rispettivamente agli insiemi (**a**) $\Theta(g(n))$, (**b**) $O(g(n))$ e (**c**) $\Omega(g(n))$. Si noti come le relazioni di diseguaglianza che compaiono nelle definizioni sono soddisfatte solo a partire da un certo valore n_0 della dimensione del dato di ingresso.

Performance

- A **sequential algorithm** is evaluated by its runtime, typically expressed as a **function of its input**
- In any case, the asymptotic running time is the same for any machine

- The **execution time** of a parallel program depends on the other hand, in addition to the **size** of the **input**, on the **number of processors** and the **communication** parameters of the machine (i.e., EVEN on the architecture type!)

- For this reason, the "rules" for the asymptotic analysis (e.g. "bubble sort algorithm is $O(n^2)$", etc.) for sequential programs are no longer **valid**: we must analyze the algorithm in the **context** of the machine that runs it

- A **parallel system** is the **combination** of a parallel algorithm and the underlying parallel platform

Performance

- Some **measures** for the performance of parallel algorithms are intuitive:

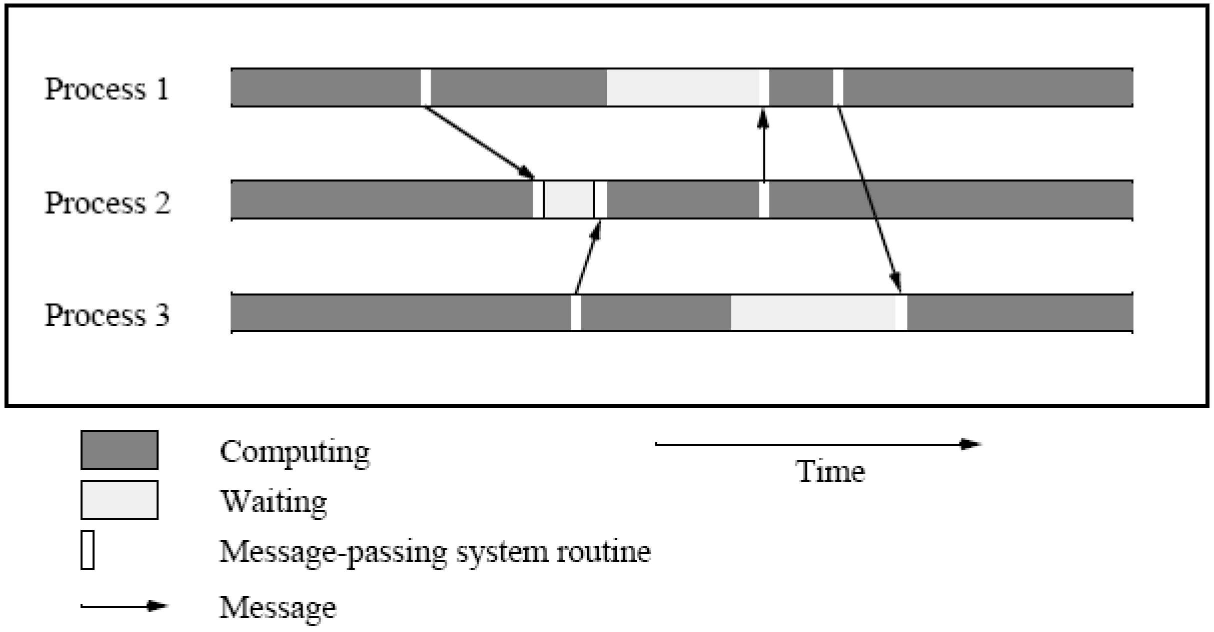
- **Wall clock time** - is the time elapsed since the start of the first processor to the final time of the last processor. What happens if you change the number of processors or if you change your architecture?

- **How much faster is our parallel version?**
- Compared with which sequential version?
  - The same algorithm on **a processor** of the parallel machine?
  - The best sequential algorithm?

Sources of Overhead in Parallel Programs

- If you use **two** processors, why does my program not run **twice** as fast?

- Unfortunately there are some "overheads" to consider, such as **excessive computation, communication and idling**



Sources of Overhead in Parallel Programs

- **Excessive computation**: Computation not performed in the serial version of the algorithm. This could be the case when we have to do with a sequential algorithm that is difficult or impossible to parallelize; therefore, we must rely on a poor efficient parallel algorithm

- **Communications**: for non-trivial problems, **INEVITABLE**!

- **Idling**: Some processes / processors may be idle due to load imbalance, synchronization or serial components (eg, I / O)

Execution times

- **Sequential Execution Time**: Elapsed time (elapsed) between the beginning and the end of the execution on a sequential computer

- **Parallel execution time**: Time elapsed since the first process starts to the moment when the last process ends

- As a rule, we denote by $\mathbf{T_s}$ the execution time of the serial, and parallel with the $\mathbf{T_P}$

- To be precise, $\mathbf{T_s(n)}$ and $\mathbf{T_p(n, p)}$ are functions, with **n** the size of the input, and **p** the number of processors

Total Parallel Overhead

- Let $\mathbf{T_{all}}$ is the total time of all processors
- Let $\mathbf{T_S}$ is the time sequential

- $\mathbf{T_{all} - T_S}$ is the total time used by all processes for non-profit work. It is defined as the total overhead T_O

- Note that

  $\mathbf{T_{all} = p\ T_P}$ (**p** is the number of processors).

- Thus:

$$\mathbf{T_O = p\ T_P - T_S}$$

Speedup

- Is the **main indicator** (preliminary) to check if the parallel algorithm is "**benefiting**" of parallelism

- The Speedup (S) is the **ratio** between the time taken to solve the problem (sequential) of a **single process** and the time taken to solve the same problem on a **parallel** computer with p identical processing elements. In most cases, the best sequential version is considered (even if it is acceptable to consider a “good” version)

$$S = T_S / T_P$$

- For example, consider the problem of **adding n numbers using n processors**

- If **n** is a power of 2, we can do this in **log n** steps, propagating the partial sums along a "logical" binary tree of processors

Speedup - Example

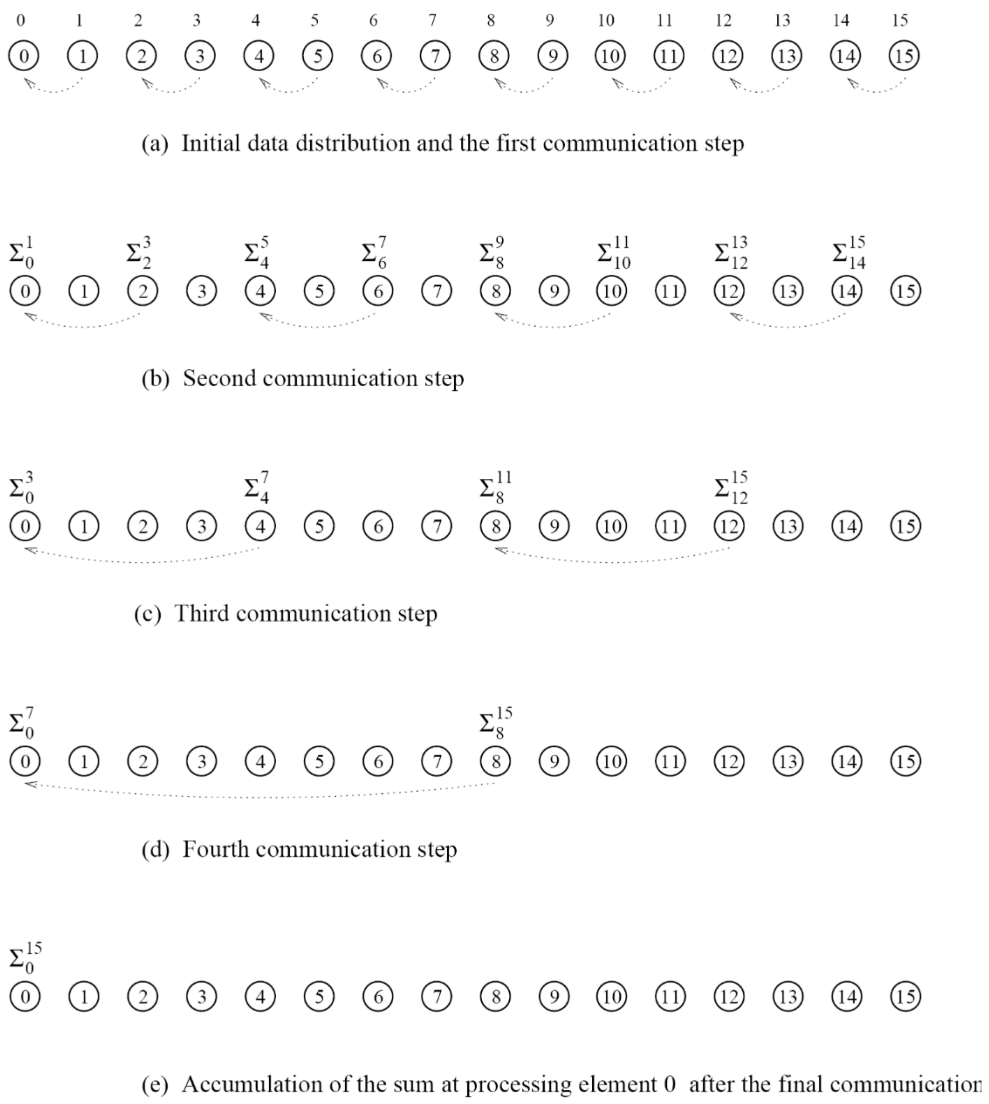

(a) Initial data distribution and the first communication step
Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}
(b) Second communication step
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(c) Third communication step
Σ_0^7 Σ_8^{15}
(d) Fourth communication step
Σ_0^{15}
(e) Accumulation of the sum at processing element 0 after the final communicatio

Figure 5.2 Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Speedup - Example

- If an addition takes a constant time, $\mathbf{t_c}$, and the communication of a single word $\mathbf{t_s} + \mathbf{t_w}$, then the parallel time is:

$$T_P = \Theta\ (\log n)$$

- We know that $T_S = \Theta\ (n)$
- Thus, the speedup S is given by $S = \Theta\ (n / \log n)$
- **NB** t_s = Startup time; t_w = Per-word transfer time

Speedup – Referred to the best sequential algorithm

- We have said that the speedup is referred to the **best sequential algorithm**. Why?

- Let’s consider an instance of **parallel bubble sort** (called "odd-even sort")

- Let the **serial time** for a bubble sort be 150 seconds
- Let the **parallel time** of odd-even sort (efficient parallelization of bubble sort) be 40 seconds

- The **speedup** seems to be 150/40 = 3.75

- Is it a fair and honest evaluation of the parallel system? Maybe not ....!

- If we take a serial quicksort of 30 seconds? In this case, the speedup is 30/40 = 0.75. **This represents a more accurate assessment of the system!**

Speedup : Greater than p?

- The minimum **speedup** can be 0 (the parallel version of the program never ends!)
- On the other hand, the **speedup** in theory, should be bounded above by p - after all, we should expect a **p-times speedup** when we use p resources!
- Speedup greater than p is possible only if each process (processor) takes less than $\boldsymbol{T_S / p}$ in solving the problem (**superlinear speedup**)
- In this case, a single processor would execute the program **faster** than the serial version, which **contradicts** our assumption that speedup refers to “the best sequential algorithm"
- **However, it is not uncommon to achieve superlinear speedup: is it good or not ?**

Speedup : Greater than p?

A possible reason is that superlinear speedup of the parallel version performs supposedly "less" work than the corresponding sequential version

Serial algorithm = 14 t_c

Parallel algorithm = 5 t_c

Speedup = 14 t_c / 5 t_c = 2.8

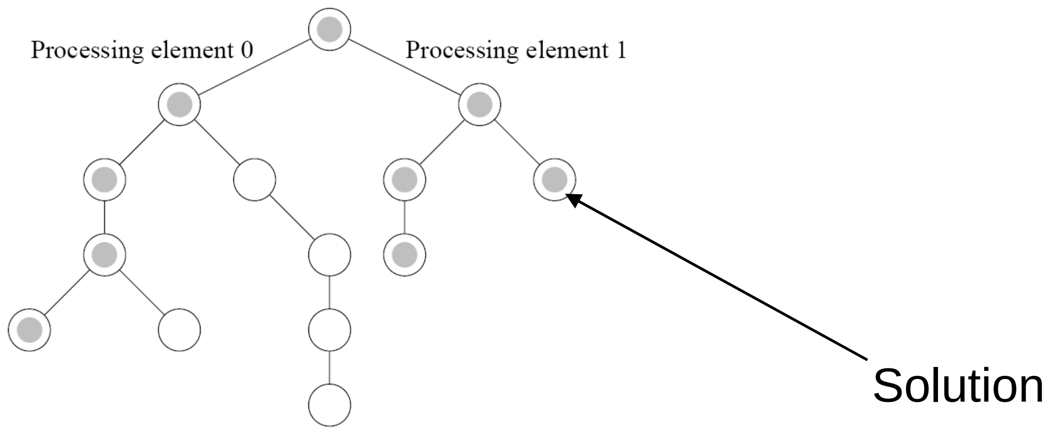


Figure 5.3 Searching an unstructured tree for a node with a given label, 'S', on two processing elements using depth-first traversal. The two-processor version with processor 0 searching the left subtree and processor 1 searching the right subtree expands only the shaded nodes before the solution is found. The corresponding serial formulation expands the entire tree. It is clear that the serial algorithm does more work than the parallel algorithm.

Speedup : Greater than p?

- **Superlinear speedup** may also be due to **hardware** reasons

- For example, data of a problem may be **too large** to be fit in the cache of a single processor, **degrading** performance because of the use of slower memory elements. However, when partitioned between processors, the **individual partitions** can be small enough to fit in their cache

- Ex: **When working with regular structures (vectors, matrices) in data-parallel problems such as (sum of elements, cellular automata, etc.)**

Efficiency

- **Efficiency** is a measure of the fraction of time in which a processing element (processor) is **effectively used**
- Mathematically :

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

- Due to the lower and upper limits of the speedup, it is a **number between 0 and 1**
- If E = 1, the program has a **linear speedup**
- If E <1 / p, the program shows a **slowdown** (i.e. "parallel computing is useless!")

Speedup - Example

- In the "sum of numbers on processors" the speedup is given by

- Efficiency is given by $S = \frac{n}{\log n}$

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

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

- **OBS**: It's not much! In fact, several processors are idle during the various sums ...

Cost of a Parallel System

- The **cost** of a parallel system is the **product** of the **parallel time** and the number of **processes** ($p \times T_P$)

- It reflects the amount of time that each process is spending in solving the problem

- The **serial cost** of a problem run on a sequential machine is trivially the **sequential runtime** (of the best algorithm)

- A parallel system is **cost-optimal** if the cost of solving a problem on a parallel machine is asymptotically (in terms of Θ), equal to the serial cost

- Since $E = T_S / p\, T_p$, we have systems for cost-optimal

$$E = \Theta(1).$$

- The **cost** is also called **work** or **processor-time product**

Cost of a Parallel System: Example

Consider the problem of the sum of **n** numbers on **n** processors

We have $\mathbf{T_P = \log n}$ (since $p = n$).

The cost of the system is given by $\mathbf{p\, T_P = n \log n}$.

Given that the sequential time is $\mathbf{\Theta(n) \neq \Theta(n \log n)}$**,** the algorithm is **not cost optimal**.

Effect of Granularity on Performance

- Often, paradoxically, the use of less processes **increases the performance** of parallel systems

- The use of less than the maximum number of processors to execute a parallel algorithm is called **scaling - down**

- A **simple way** (though impractical) to do this, called **Scaling by Emulation**, is to implement an algorithm that provides **one** input element for processor, and to subsequently use fewer processes to simulate a greater number

- If we have **n** inputs and **p** processors (p < n), we can assume **n** virtual processes and use **p** physical processors to simulate **n / p** virtual processes

Effect of Granularity on Performance

- As the number of processors is **decreased** by a factor of **n / p**, the computational load of each processor is **increased** by a factor of **n / p**. Thus, the cost (**$p \times T_p$**) does not increase, since **T_p** increases at most **n / p**

- Communication **costs** should not increase by the same factor since often the virtual processes assigned to the same processor **communicate with each other** (thus decreasing the costs actually)

- So, if the system was **cost-optimal** at the start, it will also be now the same

- **Similarly, this method will not work if the system was not initially cost-optimal.... In fact ...**

Example (Scaling by Emulation)

- Consider the problem of adding **n** numbers on **p** processors such that **p <n** and **p** and n are powers of 2

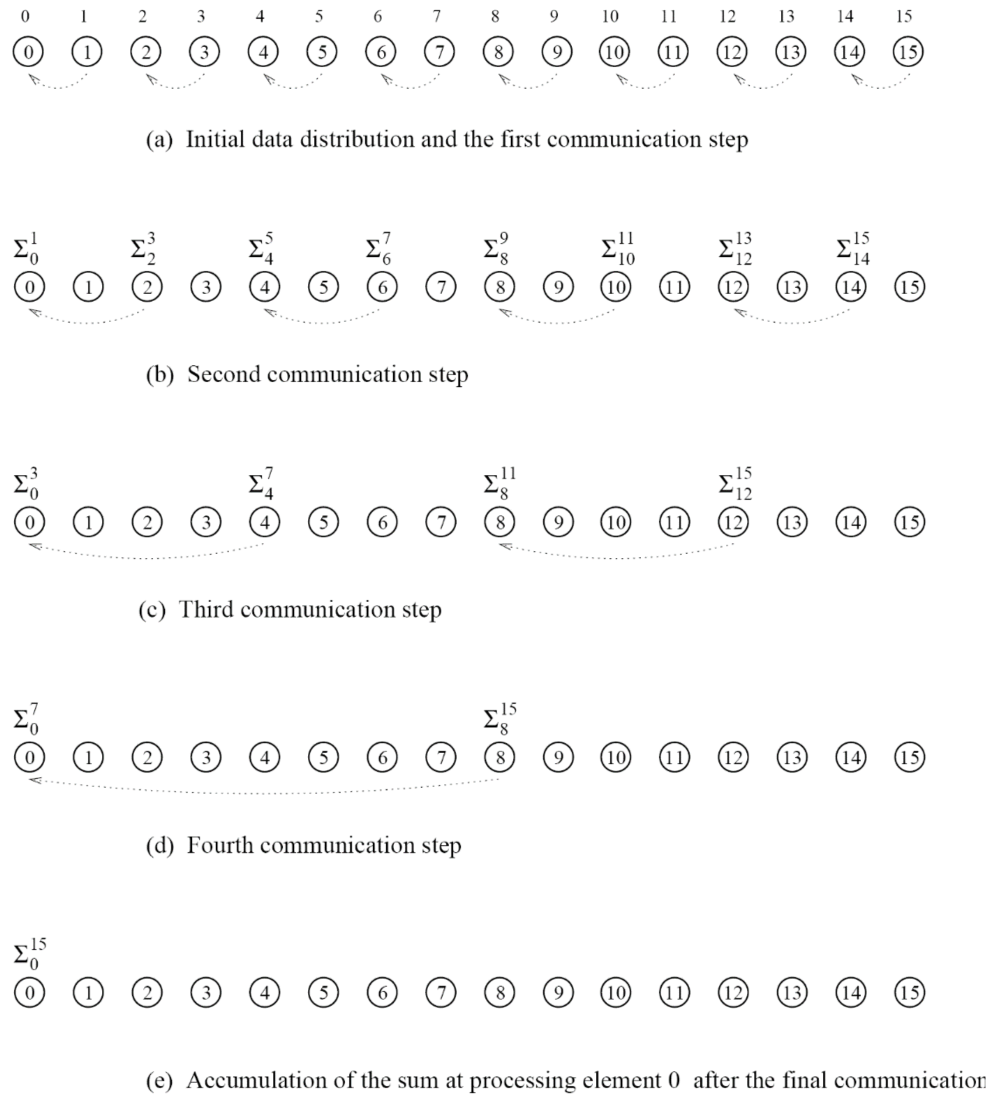
- We always use the usual algorithm for **n** processors, only now think of them as **virtual processes**

- For each processor **p** we assign **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 the “real” processors p

Speedup - Example

First 2 steps


(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

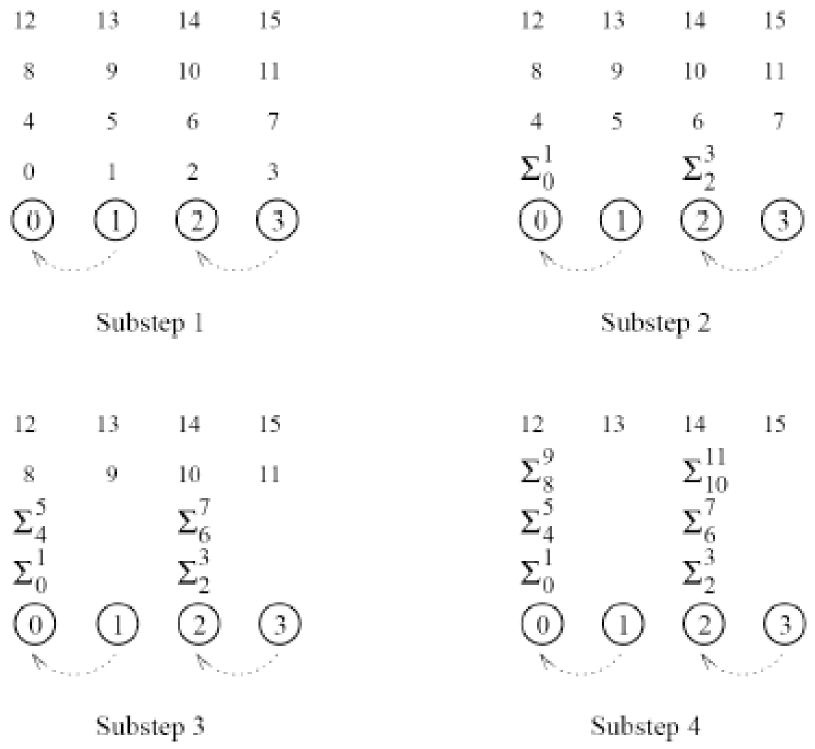
(e) Accumulation of the sum at processing element 0 after the final communicatior

Figure 5.2 Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Example

Scaling by Emulation

The first **log *p* = *(2)*** of the original algorithm are executed in (***n*** / ***p***) **log *p* = *8*** substeps that require communication



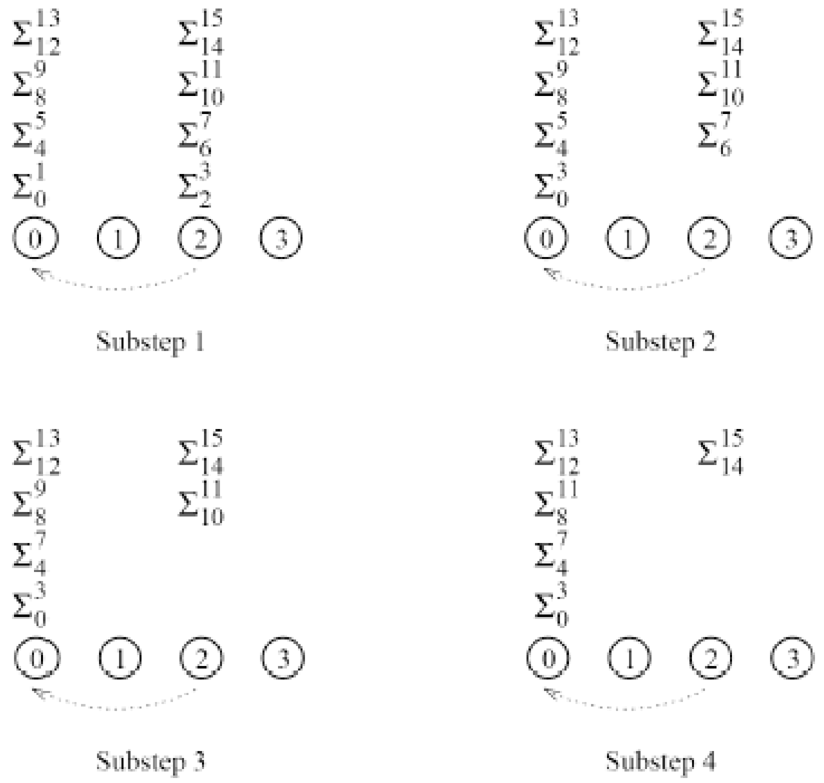
(a) Four processors simulating the first communication step of 16 processors

n **= 16**
p **= 4**

First 4 substeps…

Example (continued)

The second part of the (***n*** / ***p***) **log *p* = *8*** substeps



(b) Four processors simulating the second communication step of 16 processors

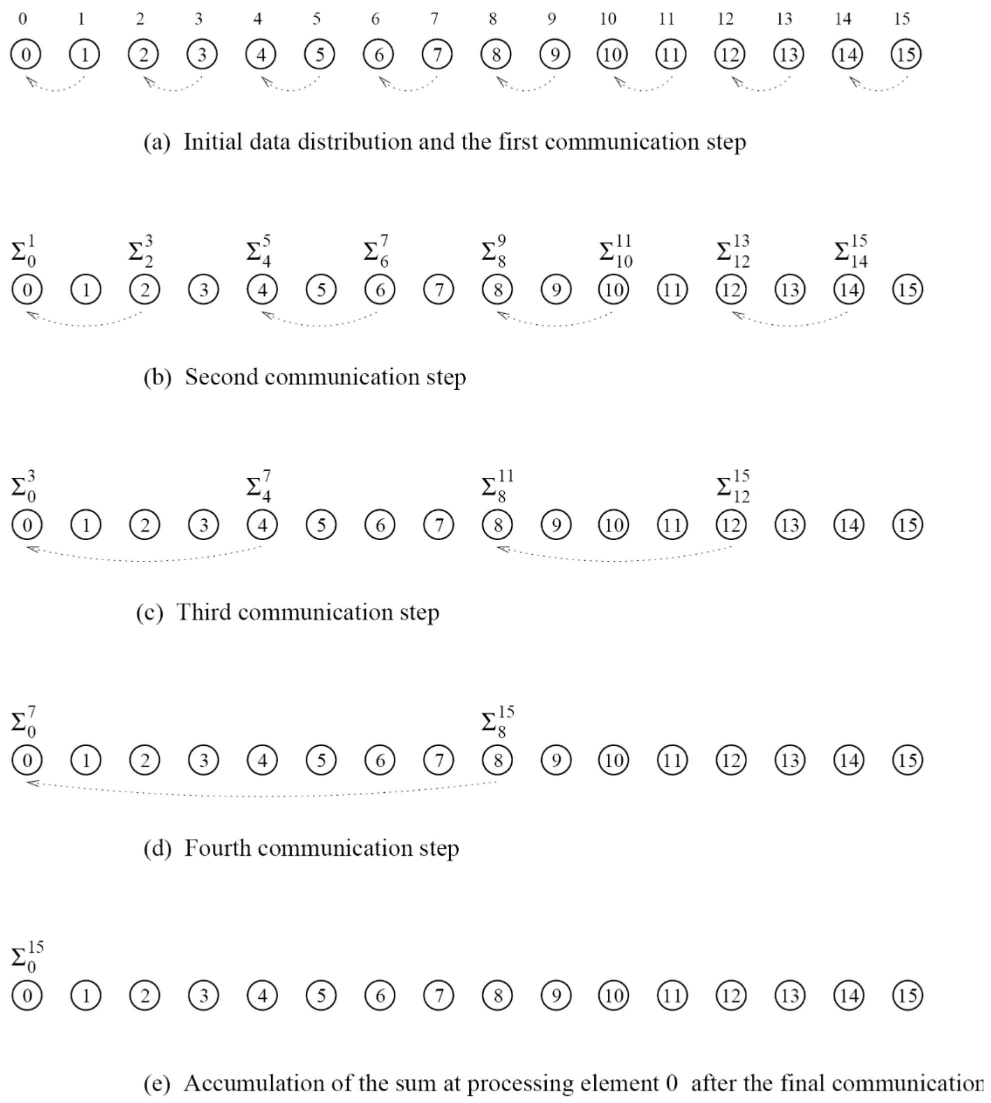
Figure 5.5 Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (first two steps). Σ_i^j denotes the sum of numbers with consecutive labels from i to j.

Remaining 4 substeps …

Example (Scaling by Emulation)

- The remaining **log n - log p** steps do not require communication (the numbers are added up "locally")
- So, the algorithm thus takes Θ ((n / p) log p) for steps that require communication, then each **single** processor must **add p / n** numbers in time Θ (n / p)
- Thus, the total parallel time is **Θ ((n / p) log p + n / p) =**
  Θ ((n / p) log p)
- As a result, the cost is Θ (n log p), asymptotically greater than the cost Θ (n) of summing the numbers sequentially. Therefore, the system is still **not cost-optimal**

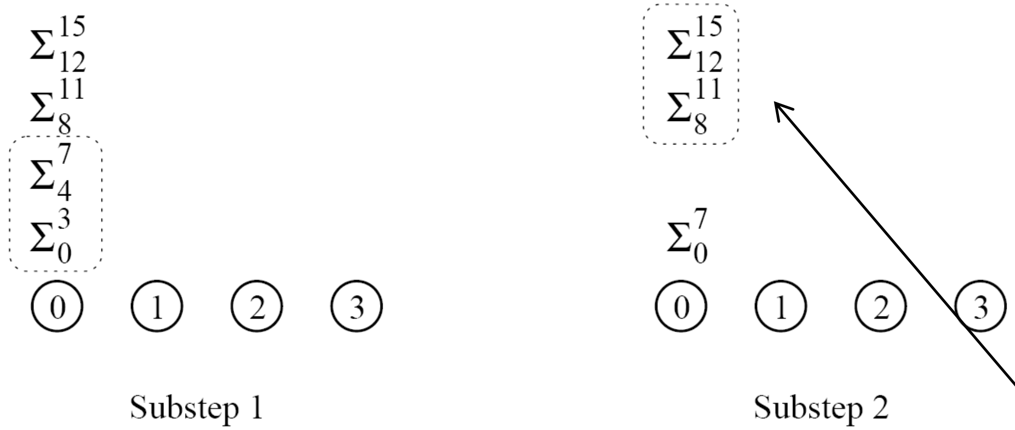
Speedup - Example



(a) Initial data distribution and the first communication step
Σ_0^1 Σ_2^3 Σ_4^5 Σ_6^7 Σ_8^9 Σ_{10}^{11} Σ_{12}^{13} Σ_{14}^{15}
(b) Second communication step
Σ_0^3 Σ_4^7 Σ_8^{11} Σ_{12}^{15}
(c) Third communication step
Σ_0^7 Σ_8^{15}
(d) Fourth communication step
Σ_0^{15}
(e) Accumulation of the sum at processing element 0 after the final communicatior

Remaining 2 steps

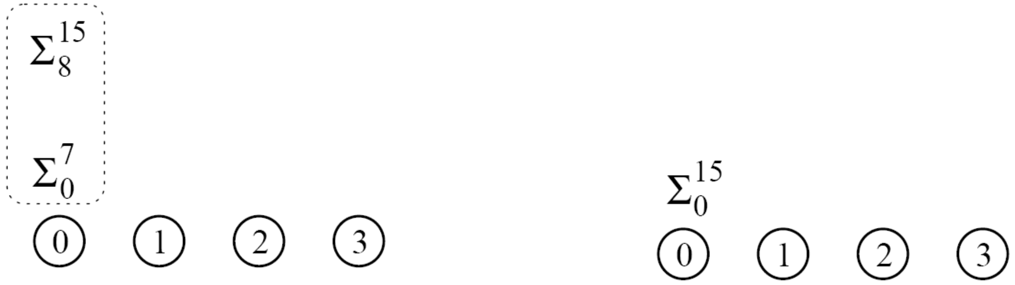
Figure 5.2 Computing the globalsum of 16 partial sums using 16 processing elements. Σ_i^j denotes the sum of numbers with consecutive labels from i to j.



The remaining
log *n* - **log *p* = *2***
steps do **<u>not require</u>** communications

(c) Simulation of the third step in two substeps

Question:
Why not add all Together ?!



(d) Simulation of the fourth step

(e) Final result

Figure 5.5 (continued) Four processing elements simulating 16 processing elements to compute the sum of 16 numbers (last three steps).

NB: **Tp**= Θ ((***n*** / ***p***) **log *p* + *n* / *p***)= Θ ((***n*** / ***p***) **log *p***)

Intelligent Scaling

- Can we "**build**" the granularity in the above example in order to obtain a **cost-optimal algorithm**?
- The previous example is **not** cost-optimal because it essentially "**simulates**" even the **communication steps** (not necessary, since we are on the same processor!)
- Another possible solution is the following (what you would do in the first place!):
  - Each processor adds up their n / p numbers in time Θ (n / p).
  - The partial sums p on p processors are added in time Θ (log p).
  - The parallel time is $T_P = \Theta(n/p + \log p)$,
  - The cost is $\Theta(n + p \log p)$

that is cost-optimal, at least until this holds:

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

Intelligent Scaling

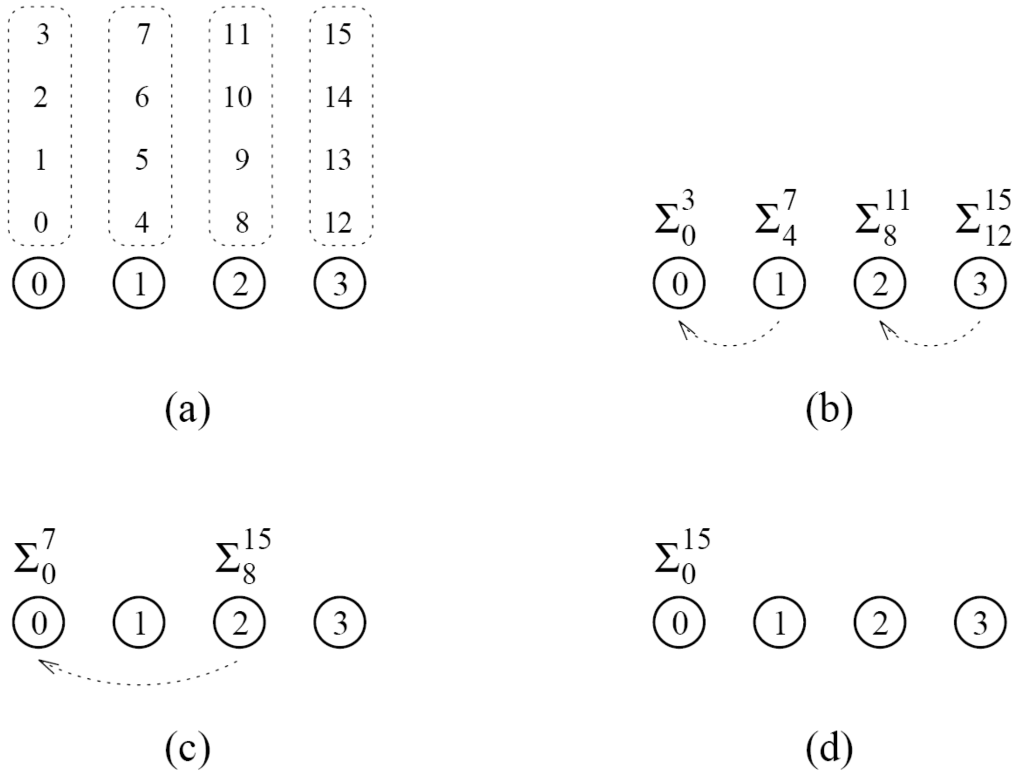


Figure 5.6 A cost-optimal way of computing the sum of 16 numbers using four processing elements.