Analytical Modeling of Parallel Systems

Dr. B Krishna Priya

Topic Overview

- Sources of Overhead in Parallel Programs
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Parallel Platforms: Models (SIMD, MIMD, SPMD) , Communication (Shared Address Space vs. Message Passing)

Analytical Modeling: Sequential Execution Time

- The execution time of a sequential algorithm
  - Asymptotic execution time as **a function of input size**
    - **identical on any serial platform**

Example: Matrix Multiplication

```
int n = A.length;                           <-- cost = c0, 1 time
for (int i = 0; i < n; i++) {               <-- cost = c1, n times
  for (int j = 0; j < n; j++) {             <-- cost = c2, n*n times
    sum = 0;                                <-- cost = c3, n*n times
    for k = 0; k < n; k++)                  <-- cost = c4, n*n*n times
      sum = sum + A[i][k]*B[k][j];          <-- cost = c5, n*n*n times
    C[i][j] = sum;                          <-- cost = c6, n*n times
  }
}
```

Total number of operations:

$= c0 + c1*n + (c2+c3+c6)*n*n + (c4+c5)*n*n*n$

$= O(n^3)$

- Big-O Notation
  - $O(1)$
  - $O(N)$
  - $O(N^2)$
  - $O(N\log N)$
  - **$O(N^3)$**
  - …

Count the number of operations

Analytical Modeling - Basics

- A **sequential algorithm** is evaluated by its runtime (in general, **asymptotic runtime** as a function of input size).
- The asymptotic runtime of a sequential program is **identical on any serial platform**.
- The **parallel runtime** of a program depends on the **input size**, the **number of processors**, and the **communication parameters** of the machine.
- An algorithm must therefore be analyzed in the context of the **underlying platform**.
- A parallel system is a combination of a parallel algorithm and an underlying platform.

Analytical Modeling - Basics

- A number of performance measures are intuitive.

- **Wall clock time** - the time from the start of the first processor to the stopping time of the last processor in a parallel ensemble. **But how does this scale when the number of processors is changed** of the program is ported to another machine altogether?

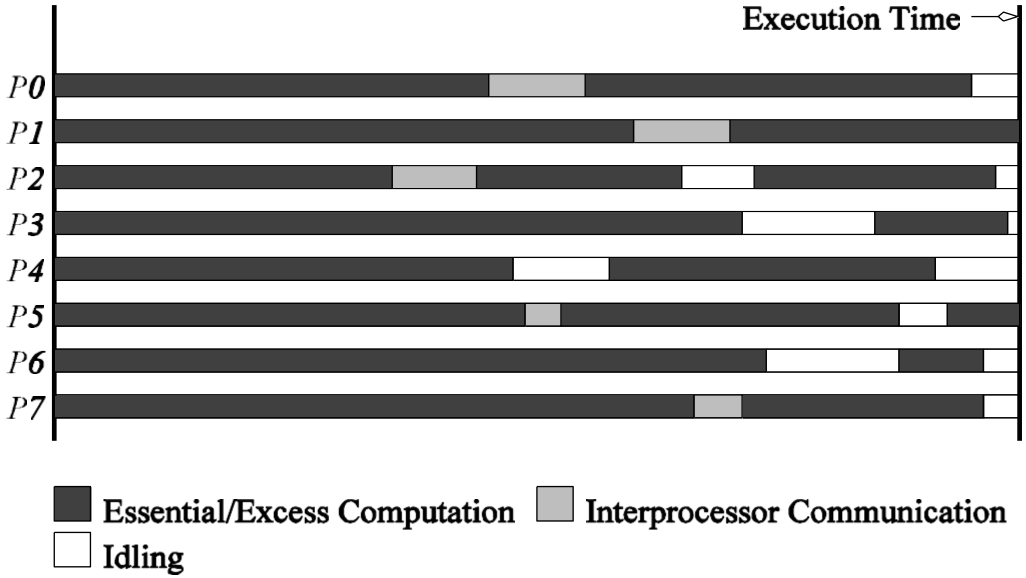
- **How much faster is the parallel version?** This begs the obvious followup question - whats the baseline serial version with which we compare? Can we use a suboptimal serial program to make our parallel program look

- **Raw FLOPS (FLoating-point Operations Per Second)** - How good is FLOPS measure when it don’t solve a problem?

Sources of Overhead in Parallel Programs

- If I use two processors, shouldnt my program run twice as fast?

- No - a number of overheads, including **wasted computation**, **communication**, **idling**, and **contention** cause degradation in performance.



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Sources of Overheads in Parallel Programs

- **Interprocess interactions**: Processors working on any non-trivial parallel problem will **need to talk to each other**.

- **Idling**: Processes may idle because of **load imbalance**, **synchronization**, or **serial components**.

- **Excess Computation**: This is computation **not performed by the serial version**. This might be because the **serial algorithm is difficult to parallelize**, or that some computations are repeated across processors **to minimize communication**.

Performance Metrics for Parallel Systems: Execution Time

- **Serial runtime** of a program is the **time elapsed between the beginning and the end** of its execution on a sequential computer.
- **The parallel runtime** is the time that elapses from the moment the **first processor starts** to the moment the **last processor finishes** execution.
- We denote the serial runtime by T_S and the parallel runtime by T_P.

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_S is the serial time.
- Observe that $T_{all} - T_S$ is then the total time spend by all processors combined in **non-useful work**. This is called the ***total overhead***.
- The total time collectively spent by all the processing elements
  $T_{all} = p\, T_P$ (p is the number of processors).
- The overhead function (T_o) is therefore given by

$$T_o = p\, T_P - T_S \qquad (1)$$

Performance Metrics for Parallel Systems: Speedup

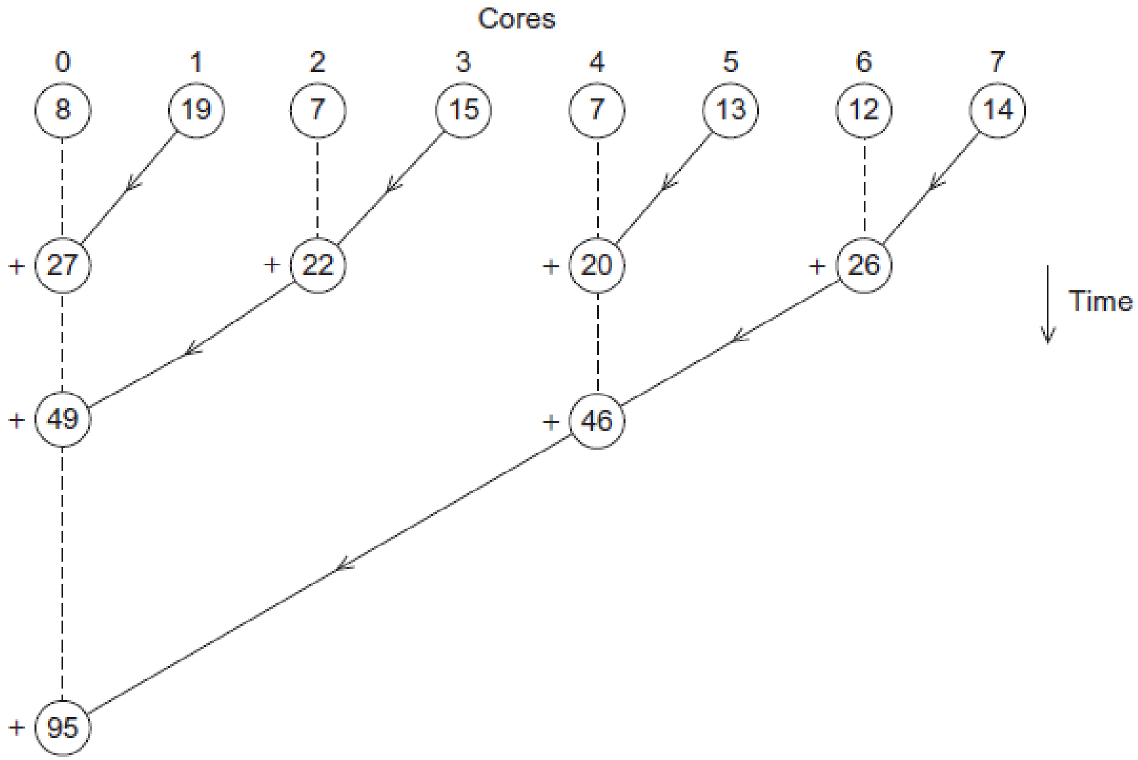
- What is the benefit from parallelism?
- **Speedup** (***S***) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

$$S = \frac{T_S}{T_P}$$

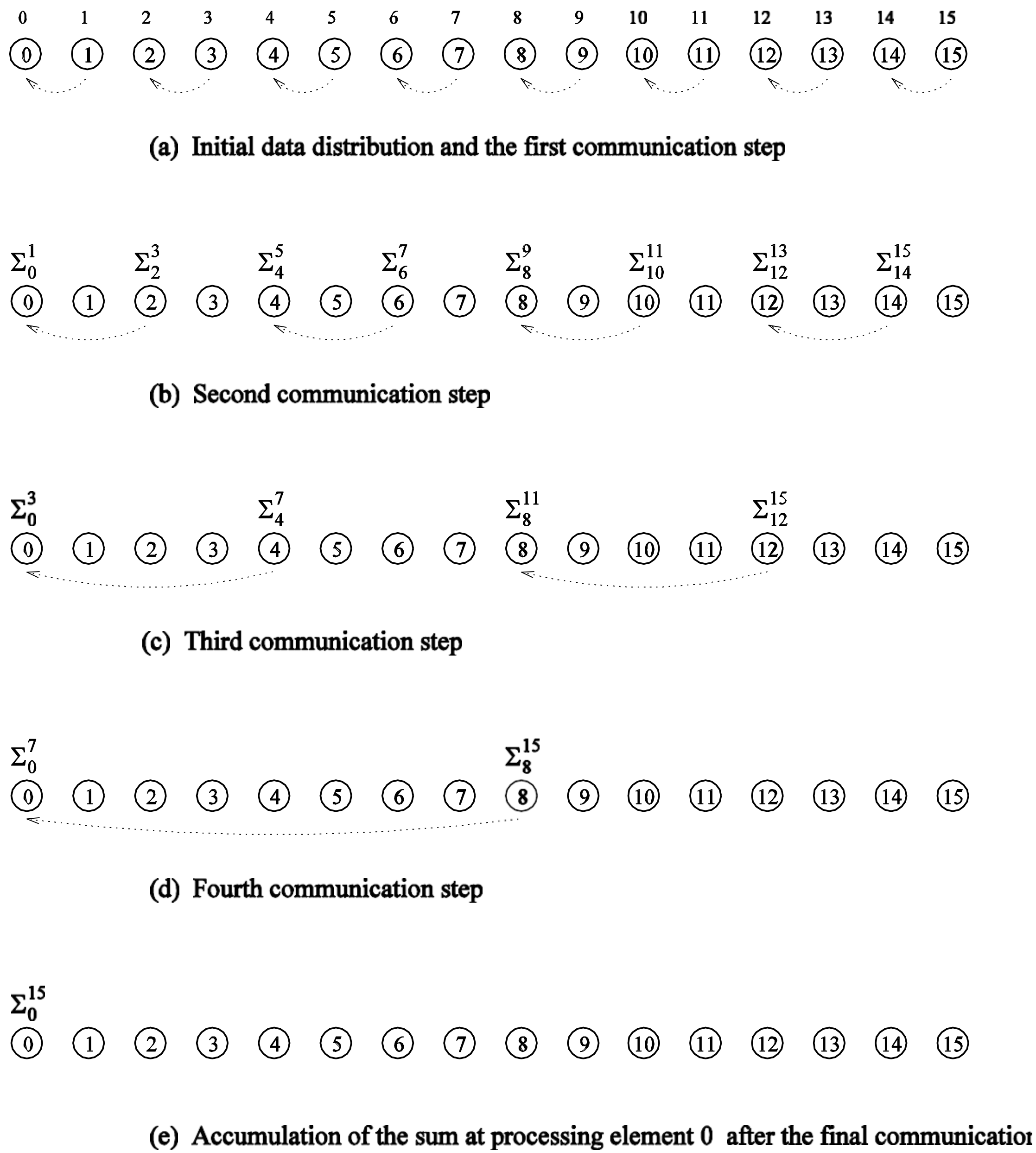
Performance Metrics: Example

- 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



(a) Initial data distribution and the first communication step

(b) Second communication step

(c) Third communication step

(d) Fourth communication step

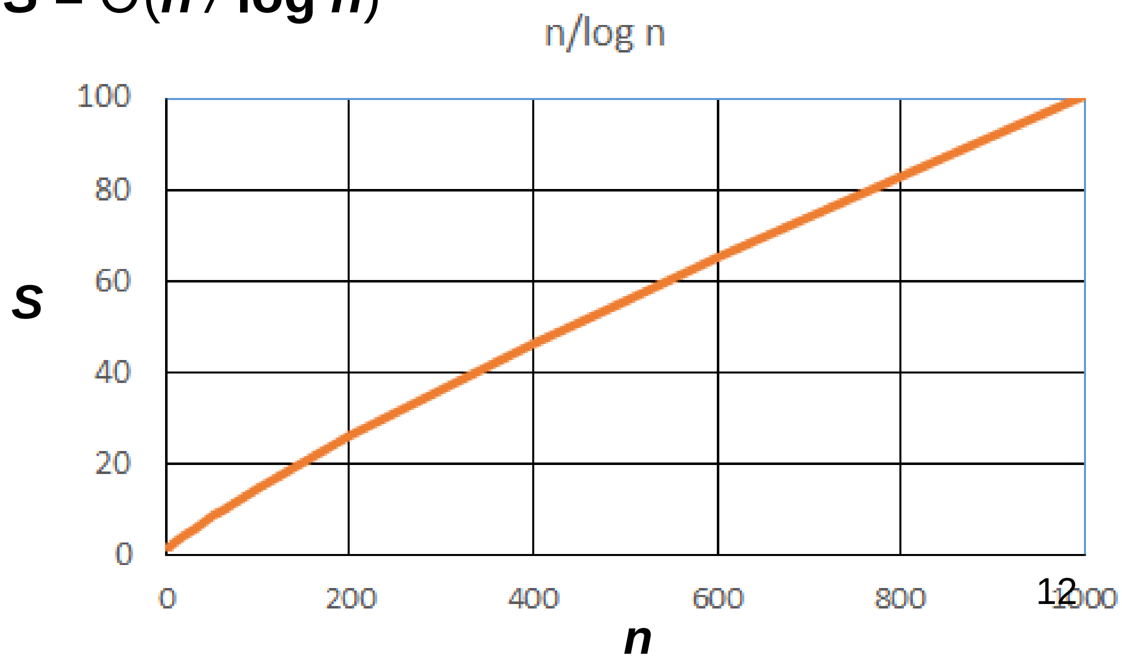
(e) Accumulation of the sum at processing element 0 after the final communicatio

Performance Metrics: Example (continued)

- If an addition takes **constant time**, say, t_c and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_P = \Theta(\log n)$

- We know that $T_S = \Theta(n)$

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



Performance Metrics: Speedup

- For a given problem, there might be **many serial algorithms** available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

- For the purpose of computing speedup, we always consider **the best sequential program as the baseline**.

Performance Metrics: Speedup Example

- Consider the problem of **parallel bubble sort**.
- The **serial time** for bubblesort is **150** seconds.
- The parallel time for **odd-even sort** (efficient parallelization of bubble sort) is **40** seconds.
- The speedup would appear to be 150/40 = **3.75**.
- But is this really a fair assessment of the system?
- What if **serial quicksort** only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system.

Performance Metrics: Speedup Bounds

- **Speedup can be as low as 0** (the parallel program never terminates).

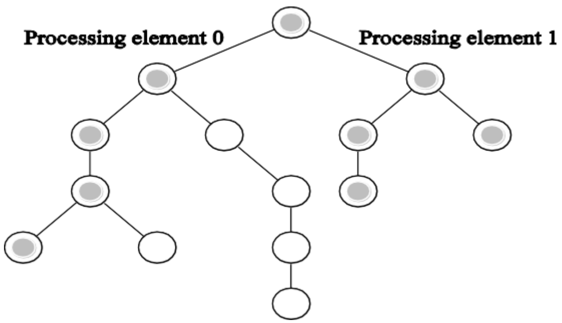
- **Speedup, in theory, should be upper bounded by *p*** - after all, we can only expect a ***p***-fold speedup if we use times as many resources.

- A **speedup greater than *p* is possible** only if each processing element spends less than time T_S / p solving the problem.

- In this case, a single processor could be timeslided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Superlinear Speedups

One reason for **superlinearity** is that the parallel version does less work than corresponding serial algorithm.



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**.

Performance Metrics: Efficiency

- **Efficiency** is a measure of the f**raction of time for which a processing element is usefully employed**

- Mathematically, it is given by

$$E = S / p = T_S / (p\, T_P) \qquad (2)$$

- Following the bounds on speedup, efficiency can be **as low as 0 and as high as 1**.

Performance Metrics: Efficiency Example

- The speedup of **adding numbers** on processors is given by

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

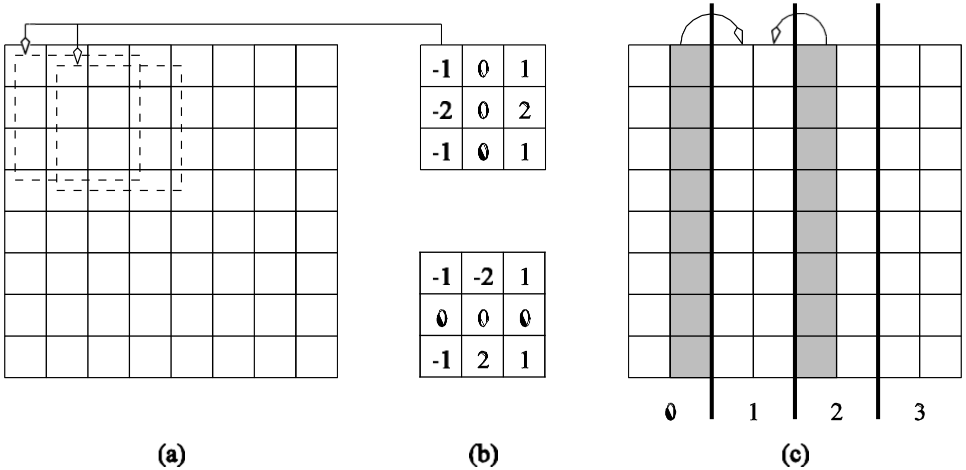
- Efficiency is given by

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

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

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 $n \times n$ image is given by $T_S = 9 t_c n^2$.



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. This takes time $2(t_s + t_w n)$.
- Templates may now be applied to all n^2 / p pixels in time $9\, t_c n^2 / p$.

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

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

Cost of a Parallel System

- **Cost** is the product of parallel runtime and the number of processing elements used ($p \times T_P$).

- 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**.

- 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*.

Cost of a Parallel System: 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**.

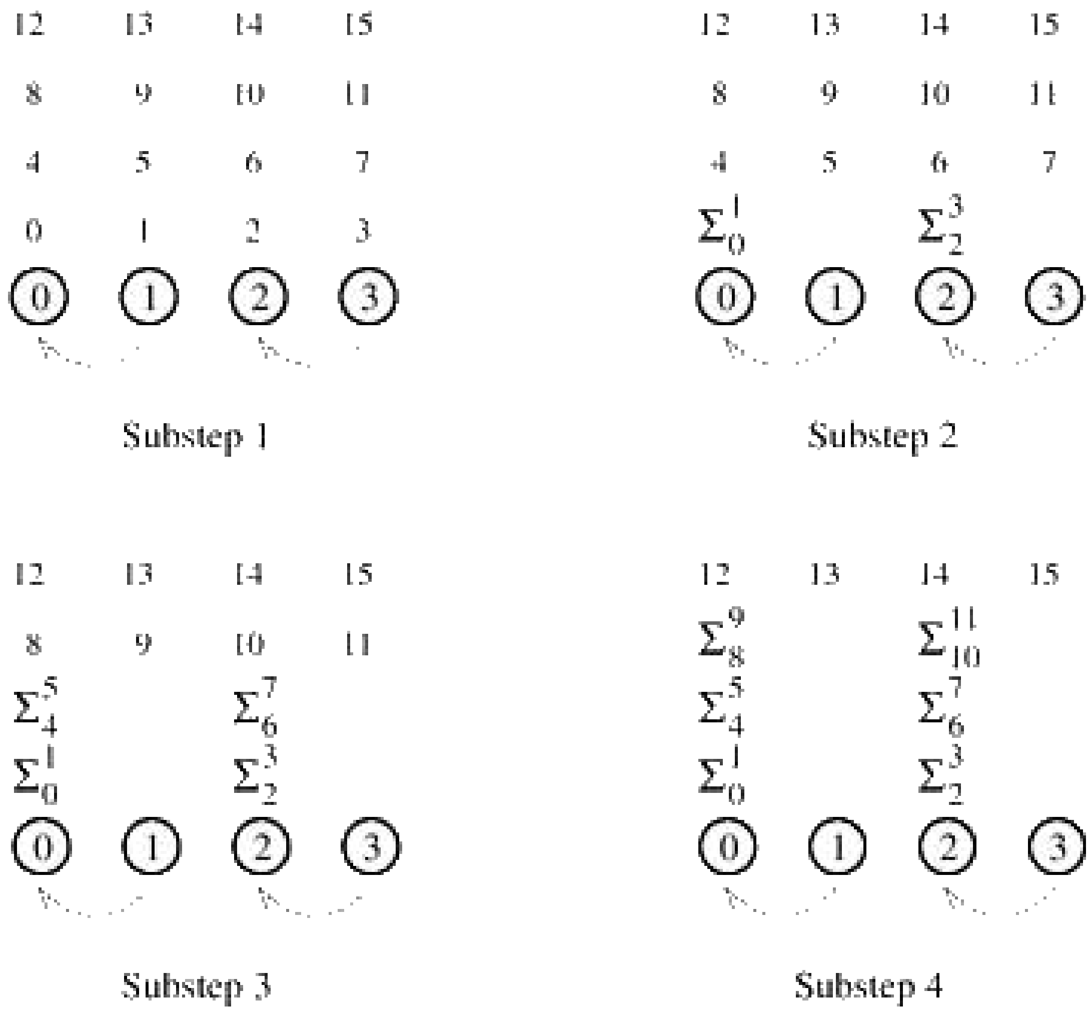
Effect of Granularity on Performance

- **Often, using fewer processors improves performance** 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 basic reason for the improvement from building granularity.

Building Granularity: Example

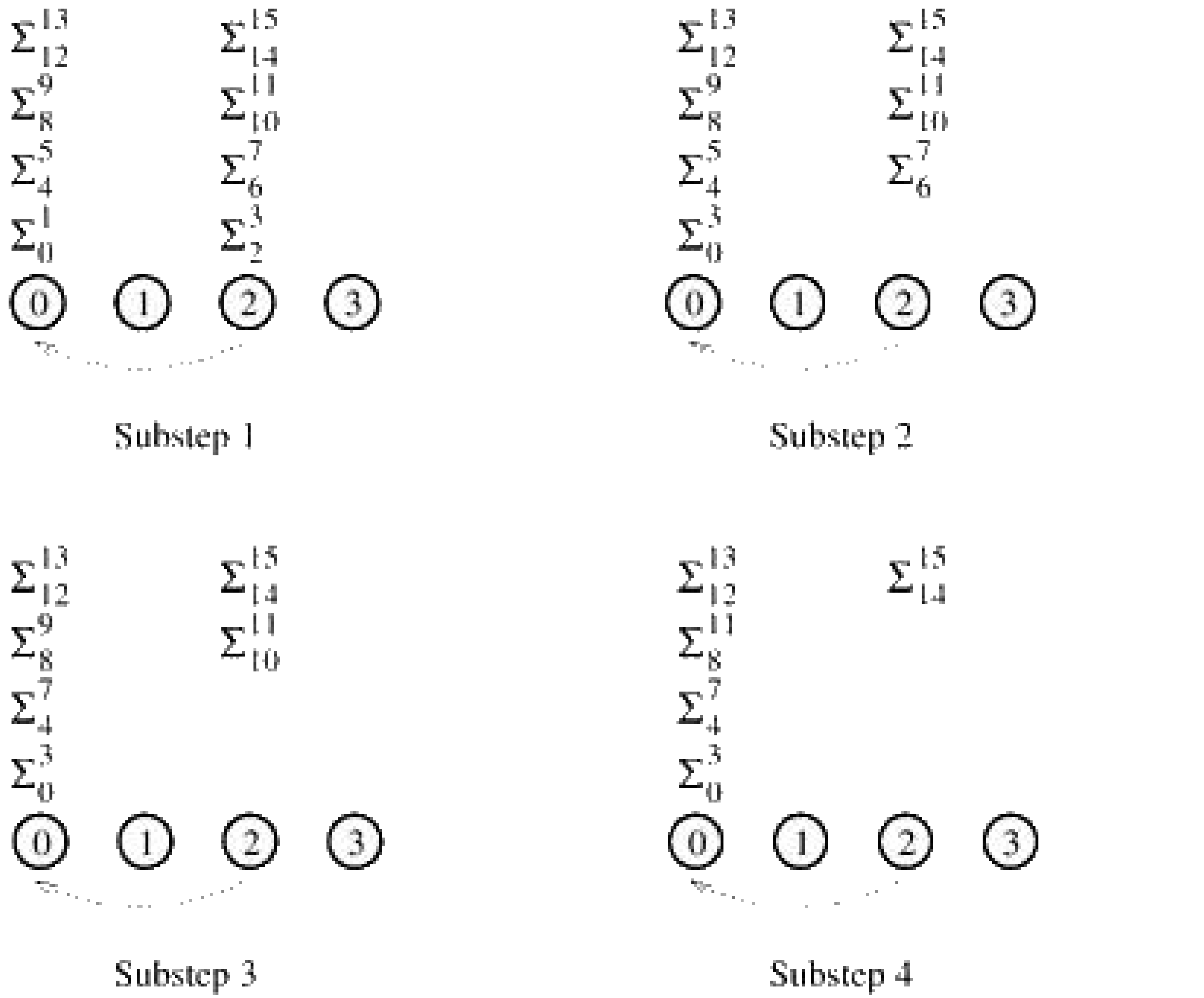
- Consider the problem of **adding *n* numbers** on ***p*** processing elements such that ***p*** < ***n*** and both ***n*** and ***p*** are powers of 2.
- Use the parallel algorithm for ***n*** processors, except, in this case, we think of them as **virtual processors**.
- Each of the ***p*** processors is now assigned ***n*** / ***p*** virtual processors.
- The first **log *p*** of 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)



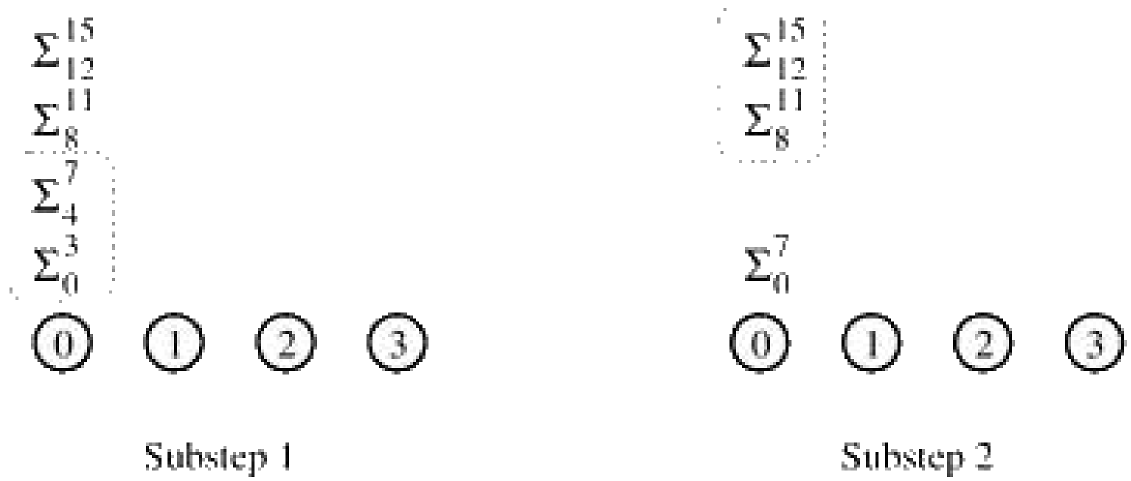
(a) Four processors simulating the first communication step of 16 processors

Building Granularity: Example (continued)

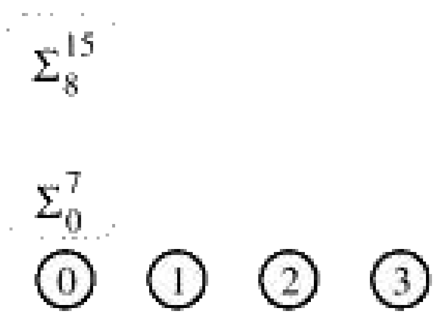


(b) Four processors simulating the second communication step of 16 processors

Building Granularity: Example (continued)



(c) Simulation of the third step in two substeps



(d) Simulation of the fourth step

(e) Final result

Building Granularity: Example (continued)

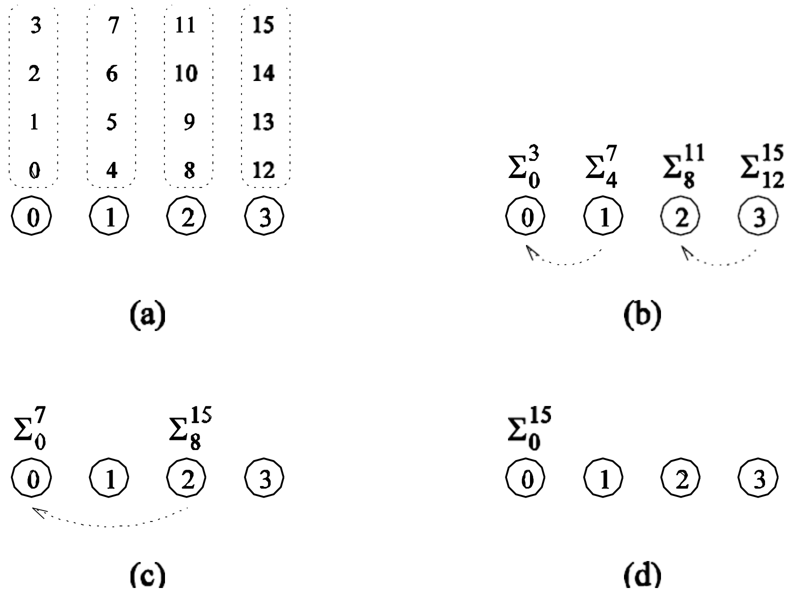
- The overall parallel execution time of this parallel system is $\Theta((\boldsymbol{n} / \boldsymbol{p}) \log \boldsymbol{p})$.

- The cost is $\Theta(\boldsymbol{n} \log \boldsymbol{p})$, which is asymptotically higher than the $\Theta(\boldsymbol{n})$ 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 $\boldsymbol{n} / \boldsymbol{p}$ numbers in time $\Theta(\boldsymbol{n} / \boldsymbol{p})$.
- The $\boldsymbol{p}$ partial sums on $\boldsymbol{p}$ processing elements can be added in time $\Theta(\log \boldsymbol{p})$.



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

$$T_P = \Theta(n/p + \log p), \qquad (3)$$

- The cost is $\Theta(n + p \log p)$
- This is **cost-optimal**, so long as $n = \Omega(p \log p)$!