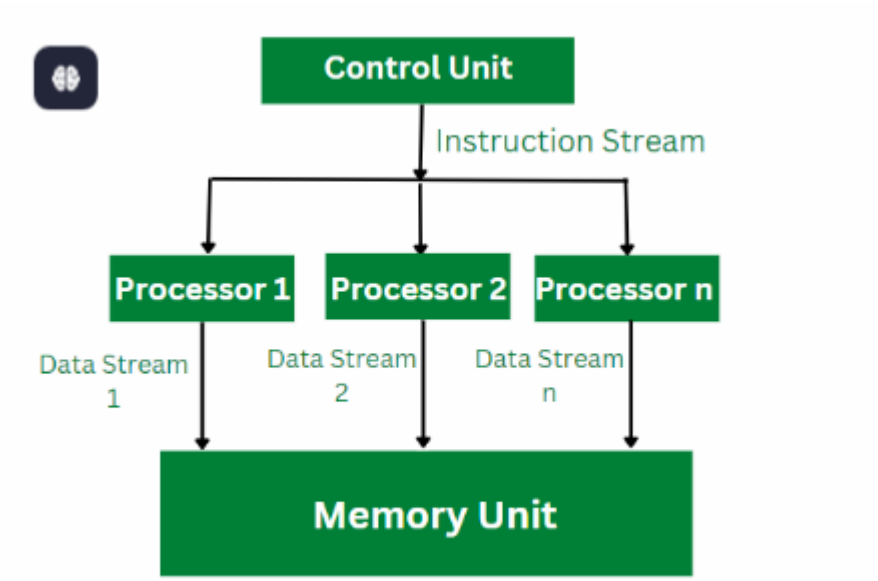
3.6 Parallel Algorithm Models

Having discussed the techniques for decomposition, mapping, and minimizing interaction overheads, we now present some of the commonly used parallel algorithm models. An algorithm model is typically a way of structuring a parallel algorithm by selecting decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

3.6.1 The Data-Parallel Model

The *data-parallel model* is one of the simplest algorithm models. In this model, the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data. This type of parallelism that is a result of identical operations being applied concurrently on different data items is called ***data parallelism***. The work may be done in phases and the data operated upon in different phases may be different. Typically, data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks. Since all tasks perform similar computations, the decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.



Dense Matrix Multiplication

Data-parallel algorithms can be implemented in both shared-address-space and message-passing paradigms. However, the partitioned address-space in a message-passing paradigm may allow better control of placement, and thus may offer a better handle on locality. On the other hand, shared-address space can ease the programming effort, especially if the distribution of data is different in different phases of the algorithm.

Interaction overheads in the data-parallel model can be minimized by choosing a locality preserving decomposition and, if applicable, by overlapping computation and interaction and by using optimized collective interaction routines. A key characteristic of data-parallel problems is that for most problems, the degree of data parallelism increases with the size of the problem, making it possible to use more processes to effectively solve larger problems.

An example of a data-parallel algorithm is dense matrix multiplication described in figure below. In the decomposition shown in below all tasks are identical; they are applied to different data.

Figure 3.10. (a) Partitioning of input and output matrices into 2 x 2 submatrices. (b) A decomposition of matrix multiplication into four tasks based on the partitioning of the matrices in (a).

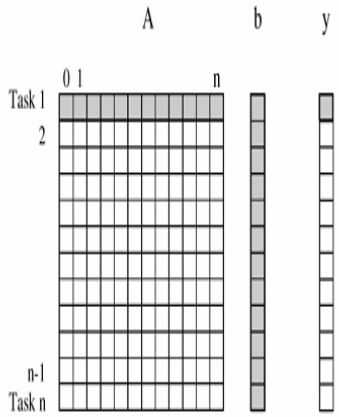
$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$
Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$
Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$
Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$

(b)

Figure 3.1. Decomposition of dense matrix-vector multiplication into *n* tasks, where *n* is the number of rows in the matrix. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



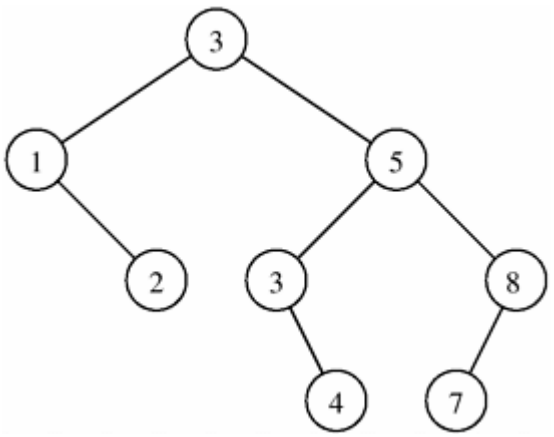
3.6.2 The Task Graph Model

The computations in any parallel algorithm can be viewed as a task-dependency graph. The task-dependency graph may be either trivial, as in the case of matrix multiplication, or nontrivial . However, in certain parallel algorithms, the task-dependency graph is explicitly used in mapping. In the *task graph model*, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs. This model is typically employed to solve problems in which the amount of data associated with the tasks is large relative to the amount of computation associated with them. Usually, tasks are mapped statically to help optimize the cost of data movement among tasks. Sometimes a decentralized dynamic mapping may be used, but even then, the mapping uses the information about the task-dependency graph structure and the interaction pattern of tasks to minimize interaction overhead. Work is more easily shared in paradigms with globally addressable space, but mechanisms are available to share work in disjoint address space.

Typical interaction-reducing techniques applicable to this model include reducing the volume and frequency of interaction by promoting locality while mapping the tasks based on the interaction pattern of tasks, and using asynchronous interaction methods to overlap the interaction with computation.

Examples of algorithms based on the task graph model include parallel quicksort), sparse matrix factorization, and many parallel algorithms derived via divide-and-conquer decomposition. This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called ***task parallelism***.

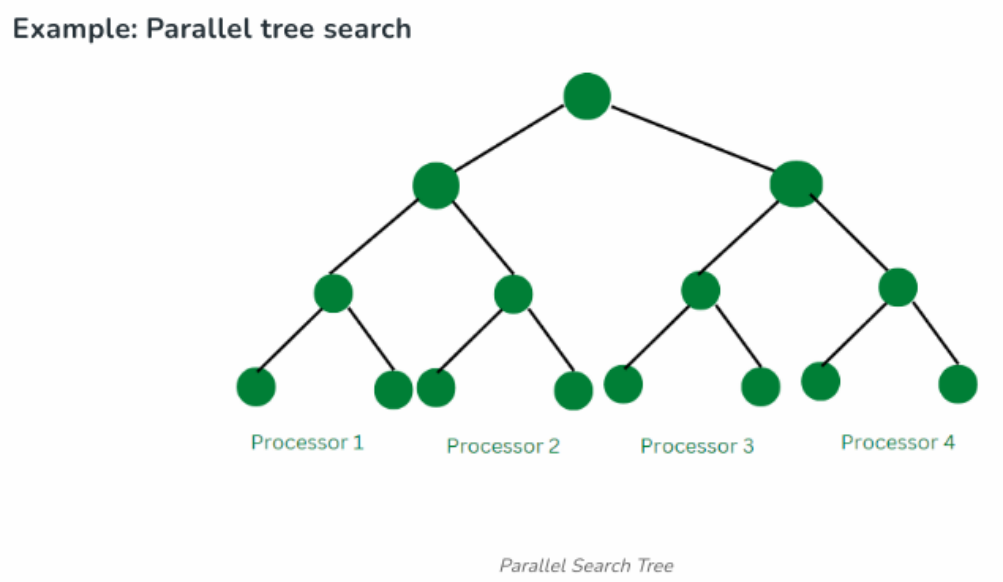
Figure 9.16. A binary tree generated by the execution of the quicksort algorithm. Each level of the tree represents a different array-partitioning iteration. If pivot selection is optimal, then the height of the tree is $\Theta(\log n)$, which is also the number of iterations.



3.6.3 The Work Pool Model

The *work pool* or the *task pool* model is characterized by a dynamic mapping of tasks onto processes for load balancing in which any task may potentially be performed by any process. There is no desired premapping of tasks onto processes. The mapping may be centralized or decentralized. Pointers to the tasks may be stored in a physically shared list, priority queue, hash table, or tree, or they could be stored in a physically distributed data structure. The work may be statically available in the beginning, or could be dynamically generated; i.e., the processes may generate work and add it to the global (possibly distributed) work pool. If the work is generated dynamically and a decentralized mapping is used, then a termination detection algorithm would be required so that all processes can actually detect the completion of the entire program (i.e., exhaustion of all potential tasks) and stop looking for more work.

In the message-passing paradigm, the work pool model is typically used when the amount of data associated with tasks is relatively small compared to the computation associated with the tasks. As a result, tasks can be readily moved around without causing too much data interaction overhead. The granularity of the tasks can be adjusted to attain the desired level of tradeoff between load-imbalance and the overhead of accessing the work pool for adding and extracting tasks.



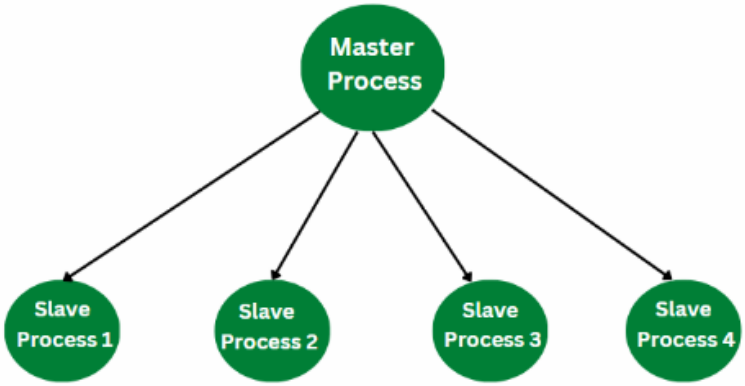
Parallel Search Tree

Parallelization of loops by chunk scheduling or related methods is an example of the use of the work pool model with centralized mapping when the tasks are statically available. Parallel tree search where the work is represented by a centralized or distributed data structure is an example of the use of the work pool model where the tasks are generated dynamically.

3.6.4 The Master-Slave Model

In the *master-slave* or the *manager-worker* model, one or more master processes generate work and allocate it to worker processes. The tasks may be allocated *a priori* if the manager can estimate the size of the tasks or if a random mapping can do an adequate job of load balancing. In another scenario, workers are assigned smaller pieces of work at different times. The latter scheme is preferred if it is time consuming for the master to generate work and hence it is not desirable to make all workers wait until the master has generated all work pieces. In some cases, work may need to be performed in phases, and work in each phase must finish before work in the next phases can be generated. In this case, the manager may cause all workers to synchronize after each phase. Usually, there is no desired premapping of work to processes, and any worker can do any job assigned to it. The manager-worker model can be generalized to the hierarchical or multi-level manager-worker model in which the top-level manager feeds large chunks of tasks to second-level managers, who further subdivide the tasks among their own workers and may perform part of the work themselves. This model is generally equally suitable to shared-address-space or message-passing paradigms since the interaction is naturally two-way; i.e., the manager knows that it needs to give out work and workers know that they need to get work from the manager.

Example: Distribution of workload across multiple slave nodes by the master process



Distribution of workload across multiple slave nodes by the master process

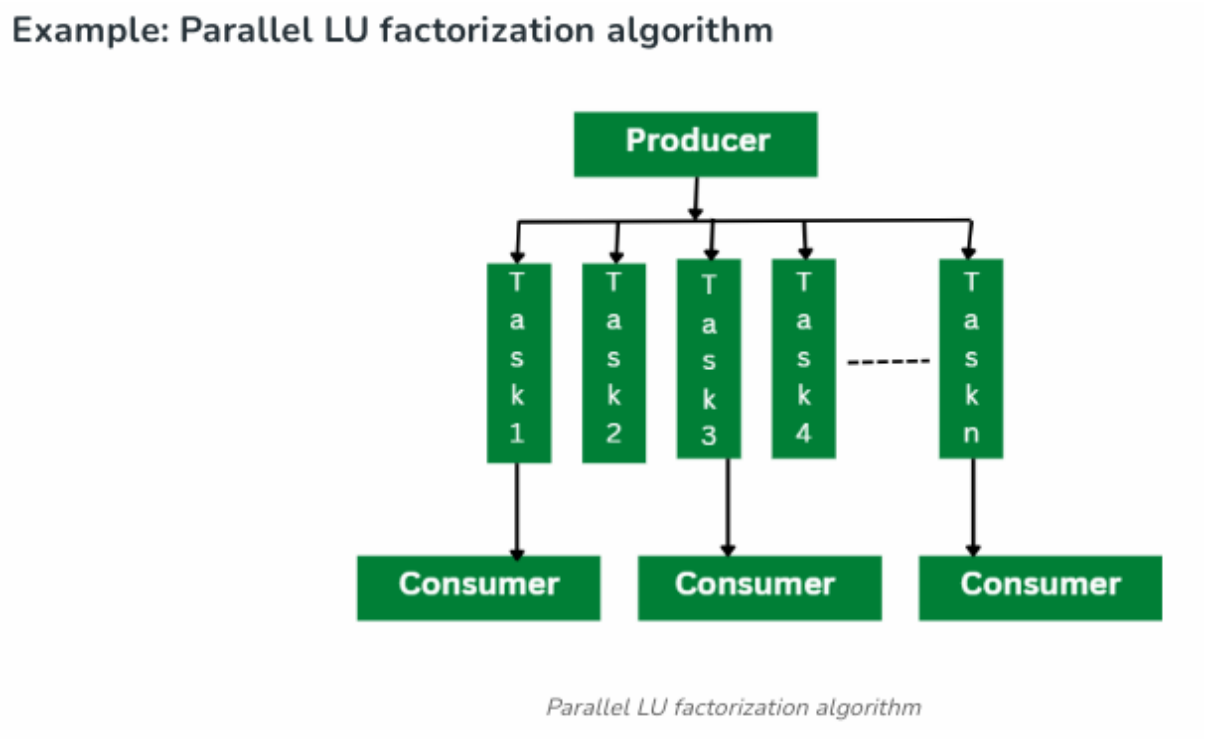
While using the master-slave model, care should be taken to ensure that the master does not become a bottleneck, which may happen if the tasks are too small (or the workers are relatively fast). The granularity of tasks should be chosen such that the cost of doing work dominates the cost of transferring work and the cost of synchronization. Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master. It may also reduce waiting times if the nature of requests from workers is non-deterministic.

3.6.5 The Pipeline or Producer-Consumer Model

In the *pipeline model*, a stream of data is passed on through a succession of processes, each of which perform some task on it. This simultaneous execution of different programs on a data stream is called ***stream parallelism***. With the exception of the process initiating the pipeline, the arrival of new data triggers the execution of a new task by a process in the pipeline. The processes could form such pipelines in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles. A pipeline is a chain of producers and consumers. Each process in the pipeline can be viewed as a consumer of a sequence of data items for the process preceding it in the pipeline and as a producer of data for the process following it in the pipeline. The pipeline does not need to be a linear chain; it can be a directed graph. The pipeline model usually involves a static mapping of tasks onto processes.

Load balancing is a function of task granularity. The larger the granularity, the longer it takes to fill up the pipeline, i.e. for the trigger produced by the first process in the chain to propagate to the last process, thereby keeping some of the processes waiting. However, too fine a granularity may increase interaction overheads because processes will need to interact to receive fresh data after smaller pieces of computation. The most common interaction reduction technique applicable to this model is overlapping interaction with computation.

Example: Parallel LU factorization algorithm



Parallel LU factorization algorithm

An example of a two-dimensional pipeline is the parallel LU factorization algorithm, which is discussed in detail in

Algorithm 8.4 A serial Gaussian elimination algorithm that converts the system of linear equations $Ax = b$ to a unit upper-triangular system $Ux = y$. The matrix U occupies the upper-triangular locations of A. This algorithm assumes that $A[k, k] \neq 0$ when it is used as a divisor on lines 6 and 7.

```
1.   procedure GAUSSIAN_ELIMINATION (A, b, y)
2.   begin
3.      for k := 0 to n - 1 do           /* Outer loop */
4.      begin
5.         for j := k + 1 to n - 1 do
6.             A[k, j] := A[k, j]/A[k, k];  /* Division step */
7.         y[k] := b[k]/A[k, k];
8.         A[k, k] := 1;
9.         for i := k + 1 to n - 1 do
10.        begin
11.           for j := k + 1 to n - 1 do
12.                 A[i, j] := A[i, j] - A[i, k] x A[k, j]; /* Elimination
step */
13.           b[i] := b[i] - A[i, k] x y[k];
14.           A[i, k] := 0;
15.        endfor;          /* Line 9 */
16.     endfor;             /* Line 3 */
17.  end GAUSSIAN_ELIMINATION
```

3.6.6 Hybrid Models

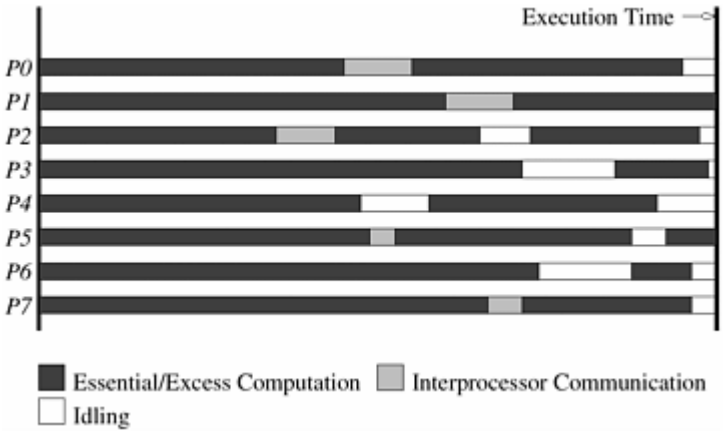
In some cases, more than one model may be applicable to the problem at hand, resulting in a hybrid algorithm model. A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm. In some cases, an algorithm formulation may have characteristics of more than one algorithm model. For instance, data may flow in a pipelined manner in a pattern guided by a task-dependency graph. In another scenario, the major computation may be described by a task-dependency graph, but each node of the graph may represent a super-task comprising multiple subtasks that may be suitable for data-parallel or pipelined parallelism. Parallel quick-sort is one of the applications for which a hybrid model is ideally suited

Sources of Overhead in Parallel Programs

Using twice as many hardware resources, one can reasonably expect a program to run twice as fast. However, in typical parallel programs, this is rarely the case, due to a variety of overheads associated with parallelism. An accurate quantification of these overheads is critical to the understanding of parallel program performance.

A typical execution profile of a parallel program is illustrated in Figure 5.1. In addition to performing essential computation (i.e., computation that would be performed by the serial program for solving the same problem instance), a parallel program may also spend time in interprocess communication, idling, and excess computation (computation not performed by the serial formulation).

Figure 5.1. 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.



Interprocess Interaction Any nontrivial parallel system requires its processing elements to interact and communicate data (e.g., intermediate results). The time spent communicating data between processing elements is usually the most significant source of parallel processing overhead.

Idling Processing elements in a parallel system may become idle due to many reasons such as load imbalance, synchronization, and presence of serial components in a program. In many parallel applications (for example, when task generation is dynamic),

it is impossible (or at least difficult) to predict the size of the subtasks assigned to various processing elements. Hence, the problem cannot be subdivided statically among the processing elements while maintaining uniform workload. If different processing elements have different workloads, some processing elements may be idle during part of the time that others are working on the problem. In some parallel programs, processing elements must synchronize at certain points during parallel program execution. If all processing elements are not ready for synchronization at the same time, then the ones that are ready sooner will be idle until all the rest are ready. Parts of an algorithm may be unparallelizable, allowing only a single processing element to work on it. While one processing element works on the serial part, all the other processing elements must wait.

Excess Computation The fastest known sequential algorithm for a problem may be difficult or impossible to parallelize, forcing us to use a parallel algorithm based on a poorer but easily parallelizable (that is, one with a higher degree of concurrency) sequential algorithm. The difference in computation performed by the parallel program and the best serial program is the excess computation overhead incurred by the parallel program.

A parallel algorithm based on the best serial algorithm may still perform more aggregate computation than the serial algorithm. An example of such a computation is the Fast Fourier Transform algorithm. In its serial version, the results of certain computations can be reused. However, in the parallel version, these results cannot be reused because they are generated by different processing elements. Therefore, some computations are performed multiple times on different processing elements. Since different parallel algorithms for solving the same problem incur varying overheads, it is important to quantify these overheads with a view to establishing a figure of merit for each algorithm.

Performance Metrics for Parallel Systems

It is important to study the performance of parallel programs with a view to determining the best algorithm, evaluating hardware platforms, and examining the benefits from parallelism. A number of metrics have been used based on the desired outcome of performance analysis.

5.2.1 Execution Time

The 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 a parallel computation starts to the moment the last processing element finishes execution. We denote the serial runtime by T_S and the parallel runtime by T_P.

5.2.2 Total Parallel Overhead

The overheads incurred by a parallel program are encapsulated into a single expression referred to as the ***overhead function***. We define overhead function or ***total overhead*** of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol T_o.

The total time spent in solving a problem summed over all processing elements is pT_P . T_S units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (T_o) is given by

Equation 5.1

$$T_o = pT_P - T_S.$$

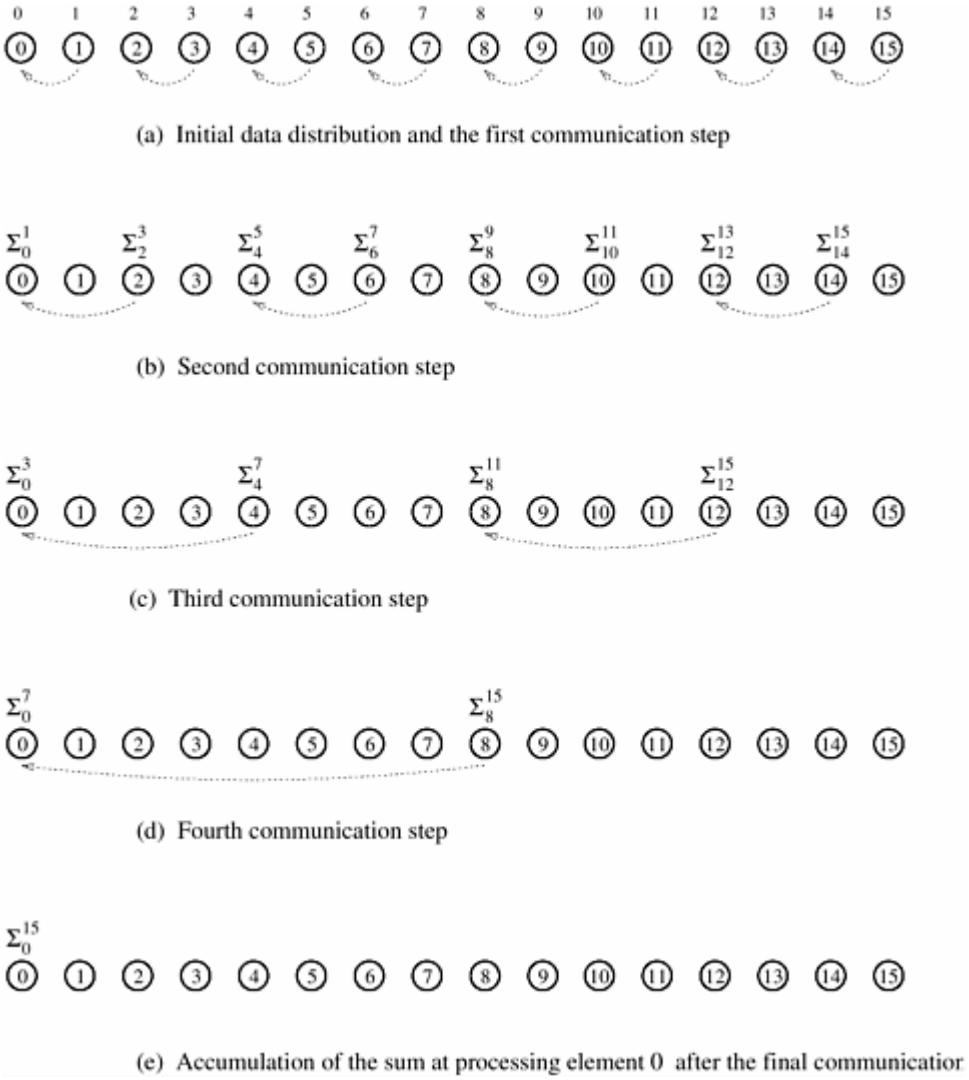
5.2.3 Speedup

When evaluating a parallel system, we are often interested in knowing how much performance gain is achieved by parallelizing a given application over a sequential implementation. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with *p* identical processing elements. We denote speedup by the symbol *S*.

Example 5.1 Adding n numbers using n processing elements

Consider the problem of adding *n* numbers by using *n* processing elements. Initially, each processing element is assigned one of the numbers to be added and, at the end of the computation, one of the processing elements stores the sum of all the numbers. Assuming that *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 processing elements. Figure 5.2 illustrates the procedure for *n* = 16. The processing elements are labeled from 0 to 15. Similarly, the 16 numbers to be added are labeled from 0 to 15. The sum of the numbers with consecutive labels from *i* to *j* is denoted by Σ_i^j .

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*.



Each step shown in Figure 5.2 consists of one addition and the communication of a single word. The addition can be performed in some constant time, say t_c, and the communication of a single word can be performed in time $t_s + t_w$. Therefore, the addition and communication operations take a constant amount of time. Thus,

Equation 5.2

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

Since the problem can be solved in $\Theta(n)$ time on a single processing element, its speedup is

Equation 5.3

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

For a given problem, more than one sequential algorithm may be available, but all of these may not be equally suitable for parallelization. When a serial computer is used, it is natural to use the sequential algorithm that solves the problem in the least amount of time. Given a parallel algorithm, it is fair to judge its performance with respect to the fastest sequential algorithm for solving the same problem on a single processing element. Sometimes, the asymptotically fastest sequential algorithm to solve a problem is not known, or its runtime has a large constant that makes it impractical to implement. In such cases, we take the fastest known algorithm that would be a practical choice for a serial computer to be the best sequential algorithm. We compare the performance of a parallel algorithm to solve a problem with that of the best sequential algorithm to solve the same problem. We formally define the ***speedup*** S as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processing elements. The p processing elements used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.

Example 5.2 Computing speedups of parallel programs

Consider the example of parallelizing bubble sort. Assume that a serial version of bubble sort of 10^5 records takes 150 seconds and a serial quick-sort can sort the same list in 30 seconds. If a parallel version of bubble sort, also called odd-even sort, takes 40 seconds on four processing elements, it would appear that the parallel odd-even sort algorithm results in a speedup of 150/40 or 3.75. However, this conclusion is misleading, as in reality the parallel algorithm results in a speedup of 30/40 or 0.75 with respect to the best serial algorithm.

Theoretically, speedup can never exceed the number of processing elements, p. If the best sequential algorithm takes T_S units of time to solve a given problem on a single processing element, then a speedup of p can be obtained on p processing elements if none of the processing elements spends more than time T_S/p. 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 processing element could emulate the p processing elements and solve the problem in fewer than T_S units of time. This is a contradiction because speedup, by definition, is computed with respect to the best sequential algorithm. If T_S is the serial runtime of the algorithm, then the problem cannot be solved in less than time T_S on a single processing element.

In practice, a speedup greater than p is sometimes observed (a phenomenon known as ***superlinear speedup***). This usually happens when the work performed by a serial algorithm is greater than its parallel formulation or due to hardware features that put the serial implementation at a disadvantage. For example, the data for a problem might be too large to fit into the cache of a single processing element, thereby degrading its performance due to the use of slower memory elements. But when partitioned among several processing elements, the individual data-partitions would be small enough to fit into their respective processing elements' caches. In the remainder of this book, we disregard superlinear speedup due to hierarchical memory.

Example 5.3 Super-linearity effects from caches

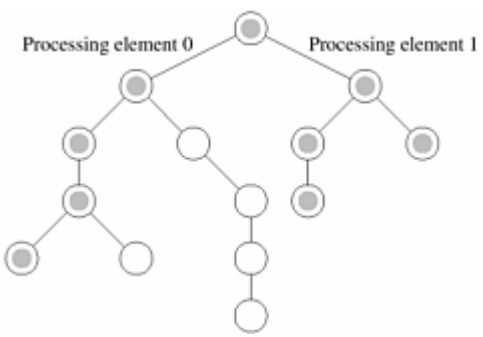
Consider the execution of a parallel program on a two-processor parallel system. The program attempts to solve a problem instance of size W. With this size and available cache of 64 KB on one processor, the program has a cache hit rate of 80%. Assuming

the latency to cache of 2 ns and latency to DRAM of 100 ns, the effective memory access time is 2 x 0.8 + 100 x 0.2, or 21.6 ns. If the computation is memory bound and performs one FLOP/memory access, this corresponds to a processing rate of 46.3 MFLOPS. Now consider a situation when each of the two processors is effectively executing half of the problem instance (i.e., size $W/2$). At this problem size, the cache hit ratio is expected to be higher, since the effective problem size is smaller. Let us assume that the cache hit ratio is 90%, 8% of the remaining data comes from local DRAM, and the other 2% comes from the remote DRAM (communication overhead). Assuming that remote data access takes 400 ns, this corresponds to an overall access time of 2 x 0.9 + 100 x 0.08 + 400 x 0.02, or 17.8 ns. The corresponding execution rate at each processor is therefore 56.18, for a total execution rate of 112.36 MFLOPS. The speedup in this case is given by the increase in speed over serial formulation, i.e., 112.36/46.3 or 2.43! Here, because of increased cache hit ratio resulting from lower problem size per processor, we notice superlinear speedup. ■

Example 5.4 Super-linearity effects due to exploratory decomposition

Consider an algorithm for exploring leaf nodes of an unstructured tree. Each leaf has a label associated with it and the objective is to find a node with a specified label, in this case 'S'. Such computations are often used to solve combinatorial problems, where the label 'S' could imply the solution to the problem. In Figure 5.3, we illustrate such a tree. The solution node is the rightmost leaf in the tree. A serial formulation of this problem based on depth-first tree traversal explores the entire tree, i.e., all 14 nodes. If it takes time t_c to visit a node, the time for this traversal is $14t_c$. Now consider a parallel formulation in which the left subtree is explored by processing element 0 and the right subtree by processing element 1. If both processing elements explore the tree at the same speed, the parallel formulation explores only the shaded nodes before the solution is found. Notice that the total work done by the parallel algorithm is only nine node expansions, i.e., $9t_c$. The corresponding parallel time, assuming the root node expansion is serial, is $5t_c$ (one root node expansion, followed by four node expansions by each processing element). The speedup of this two-processor execution is therefore $14t_c/5t_c$, or 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.



The cause for this superlinearity is that the work performed by parallel and serial algorithms is different. Indeed, if the two-processor algorithm was implemented as two processes on the same processing element, the algorithmic superlinearity would disappear for this problem instance. Note that when exploratory decomposition is used, the relative amount of work performed by serial and parallel algorithms is dependent upon the location of the solution, and it is often not possible to find a serial algorithm that is optimal for all instances.

5.2.4 Efficiency

Only an ideal parallel system containing p processing elements can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw in Example 5.1, part of the time required by the processing elements to compute the sum of n numbers is spent idling (and communicating in real systems). ***Efficiency*** is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by

Equation 5.4

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

Example 5.5 Efficiency of adding n numbers on n processing elements

From Equation 5.3 and the preceding definition, the efficiency of the algorithm for adding n numbers on n processing elements is

$$\begin{aligned} E &= \frac{\Theta\left(\frac{n}{\log n}\right)}{n} \\ &= \Theta\left(\frac{1}{\log n}\right) \end{aligned}$$

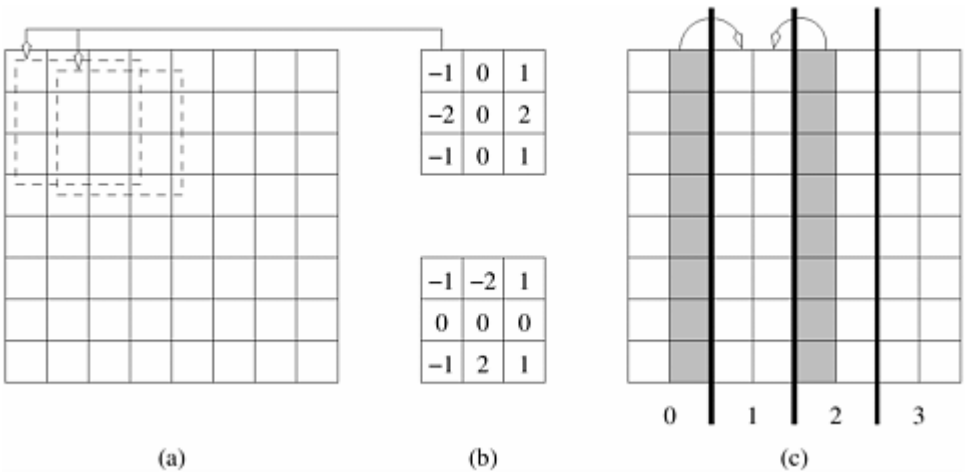
■

We also illustrate the process of deriving the parallel runtime, speedup, and efficiency while preserving various constants associated with the parallel platform.

Example 5.6 Edge detection on images

Given an n x n pixel image, the problem of detecting edges corresponds to applying a3x 3 template to each pixel. The process of applying the template corresponds to multiplying pixel values with corresponding template values and summing across the template (a convolution operation). This process is illustrated in Figure 5.4(a) along with typical templates (Figure 5.4(b)). Since we have nine multiply-add operations for each pixel, if each multiply-add takes time t_c, the entire operation takes time $9t_cn^2$ on a serial computer.

Figure 5.4. 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.



A simple parallel algorithm for this problem partitions the image equally across the processing elements and each processing element applies the template to its own sub image. Note that for applying the template to the boundary pixels, a processing element must get data that is assigned to the adjoining processing element. Specifically, if a processing element is assigned a vertically sliced sub-image of dimension n x (n/p), it must access a single layer of n pixels from the processing element to the left and a single layer of n pixels from the processing element to the right (note that one of these accesses is redundant for the two processing elements assigned the sub-images at the extremities). This is illustrated in Figure 5.4(c).

On a message passing machine, the algorithm executes in two steps: (i) exchange a layer of n pixels with each of the two adjoining processing elements; and (ii) apply template on local subimage. The first step involves two n-word messages (assuming each pixel takes a word to communicate RGB data). This takes time $2(t_s + t_wn)$. The second step takes time $9t_cn^2/p$. The total time for the algorithm is therefore given by:

$$T_P = 9t_c\frac{n^2}{p} + 2(t_s + t_wn)$$

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

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

5.2.5 Cost

We define the ***cost*** of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processing elements.

The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be ***cost-optimal*** if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Θ terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of $\Theta(1)$.

Cost is sometimes referred to as ***work*** or ***processor-time product***, and a cost-optimal system is also known as a pT_P -optimal system.

Example 5.7 Cost of adding n numbers on n processing elements

The algorithm given in Example 5.1 for adding n numbers on n processing elements has a processor-time product of $\Theta(n \log n)$. Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal. ■

Cost optimality is a very important practical concept although it is defined in terms of asymptotics. We illustrate this using the following example.

Example 5.8 Performance of non-cost optimal algorithms

Consider a sorting algorithm that uses n processing elements to sort the list in time $(\log n)^2$. Since the serial runtime of a (comparison-based) sort is $n \log n$, the speedup and efficiency of this algorithm are given by $n/\log n$ and $1/\log n$, respectively. The pT_P product of this algorithm is $n(\log n)^2$. Therefore, this algorithm is not cost optimal but only by a factor of $\log n$. Consider the problem of sorting 1024 numbers (n = 1024, $\log n$ = 10) on 32 processing elements. The speedup expected is only $p/\log n$ or 3.2. This number gets worse as n increases. For $n = 10^6$, $\log n = 20$ and the speedup is only 1.6. Clearly, there is a significant cost associated with not being cost-optimal even by a very small factor (note that a factor of $\log p$ is smaller than even $\sqrt{p}$). This emphasizes the practical importance of cost-optimality.