Chapter 3. Principles of Parallel Algorithm Design

Algorithm development is a critical component of problem solving using computers. A sequential algorithm is essentially a recipe or a sequence of basic steps for solving a given problem using a serial computer. Similarly, a parallel algorithm is a recipe that tells us how to solve a given problem using multiple processors. However, specifying a parallel algorithm involves more than just specifying the steps. At the very least, a parallel algorithm has the added dimension of concurrency and the algorithm designer must specify sets of steps that can be executed simultaneously. This is essential for obtaining any performance benefit from the use of a parallel computer. In practice, specifying a nontrivial parallel algorithm may include some or all of the following:

- Identifying portions of the work that can be performed concurrently.
- Mapping the concurrent pieces of work onto multiple processes running in parallel.
- Distributing the input, output, and intermediate data associated with the program.
- Managing accesses to data shared by multiple processors.
- Synchronizing the processors at various stages of the parallel program execution.

Typically, there are several choices for each of the above steps, but usually, relatively few combinations of choices lead to a parallel algorithm that yields performance commensurate with the computational and storage resources employed to solve the problem. Often, different choices yield the best performance on different parallel architectures or under different parallel programming paradigms.

In this chapter, we methodically discuss the process of designing and implementing parallel algorithms. We shall assume that the onus of providing a complete description of a parallel algorithm or program lies on the programmer or the algorithm designer. Tools and compilers for automatic parallelization at the current state of the art seem to work well only for highly structured programs or portions of programs. Therefore, we do not consider these in this chapter or elsewhere in this book.

3.1 Preliminaries

Dividing a computation into smaller computations and assigning them to different processors for parallel execution are the two key steps in the design of parallel algorithms. In this section, we present some basic terminology and introduce these two key steps in parallel algorithm design using matrix-vector multiplication and database query processing as examples.

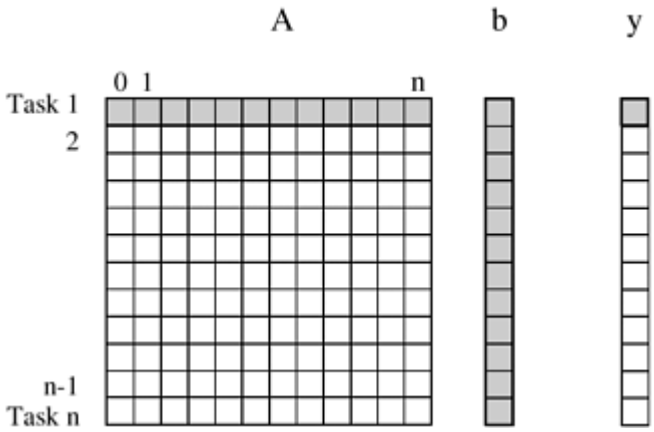
3.1.1 Decomposition, Tasks, and Dependency Graphs

The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called *decomposition*. *Tasks* are programmer-defined units of computation into which the main computation is subdivided by means of decomposition. Simultaneous execution of multiple tasks is the key to reducing the time required to solve the entire problem. Tasks can be of arbitrary size, but once defined, they are regarded as indivisible units of computation. The tasks into which a problem is decomposed may not all be of the same size.

Example 3.1 Dense matrix-vector multiplication

Consider the multiplication of a dense $n \times n$ matrix A with a vector b to yield another vector y. The ith element $y[i]$ of the product vector is the dot-product of the ith row of A with the input vector b; i.e., $y[i] = \sum_{j=1}^{n} A[i, j].b[j]$. As shown later in Figure 3.1, the computation of each $y[i]$ can be regarded as a task. Alternatively, as shown later in Figure 3.4, the computation could be decomposed into fewer, say four, tasks where each task computes roughly $n/4$ of the entries of the vector y. ■

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.



Note that all tasks in Figure 3.1 are independent and can be performed all together or in any sequence. However, in general, some tasks may use data produced by other tasks and thus may need to wait for these tasks to finish execution. An abstraction used to express such dependencies among tasks and their relative order of execution is known as a *task-dependency graph*. A task-dependency graph is a directed acyclic graph in which the nodes represent tasks and the directed edges indicate the dependencies amongst them. The task corresponding to a node can be executed when all tasks connected to this node by incoming edges have completed. Note that task-dependency graphs can be disconnected and the edge-set of a task-dependency graph can be empty. This is the case for matrix-vector multiplication, where each task computes a subset of the entries of the product vector. To see a more interesting task-dependency graph, consider the following database query processing example.

Example 3.2 Database query processing

Table 3.1 shows a relational database of vehicles. Each row of the table is a record that contains data corresponding to a particular vehicle, such as its ID, model, year, color, etc. in various fields. Consider the computations performed in processing the following query:

```
MODEL="Civic" AND YEAR="2001" AND (COLOR="Green" OR COLOR="White")
```

This query looks for all 2001 Civics whose color is either Green or White. On a relational database, this query is processed by creating a number of intermediate tables. One possible way is to first create the following four tables: a table containing all Civics, a table containing all 2001-model cars, a table containing all green-colored cars, and a table containing all white-colored cars. Next, the computation proceeds by combining these tables by computing their pairwise intersections or unions. In particular, it computes the intersection of the Civic-table with the 2001-model year table, to construct a table of all 2001-model Civics. Similarly, it computes the union of the green- and white-colored tables to compute a table storing all cars whose color is either green or white. Finally, it computes the intersection of the table containing all the 2001 Civics with the table containing all the green or white vehicles, and returns the desired list. ■

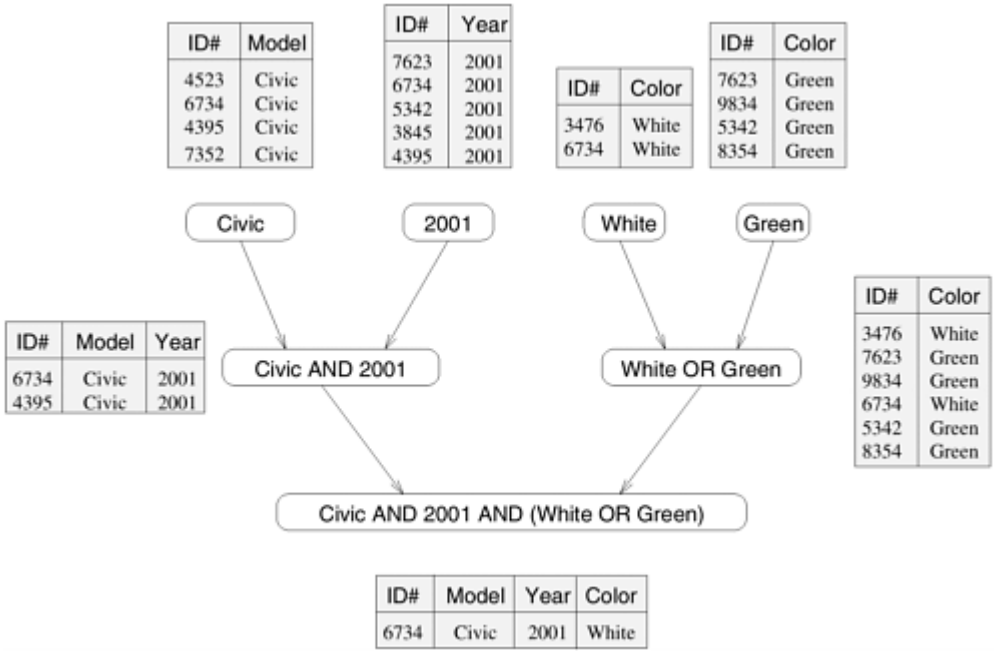
Table 3.1. A database storing information about used vehicles.

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	$18,000
3476	Corolla	1999	White	IL	$15,000
7623	Camry	2001	Green	NY	$21,000
9834	Prius	2001	Green	CA	$18,000
6734	Civic	2001	White	OR	$17,000
5342	Altima	2001	Green	FL	$19,000

ID#	Model	Year	Color	Dealer	Price
3845	Maxima	2001	Blue	NY	$22,000
8354	Accord	2000	Green	VT	$18,000
4395	Civic	2001	Red	CA	$17,000
7352	Civic	2002	Red	WA	$18,000

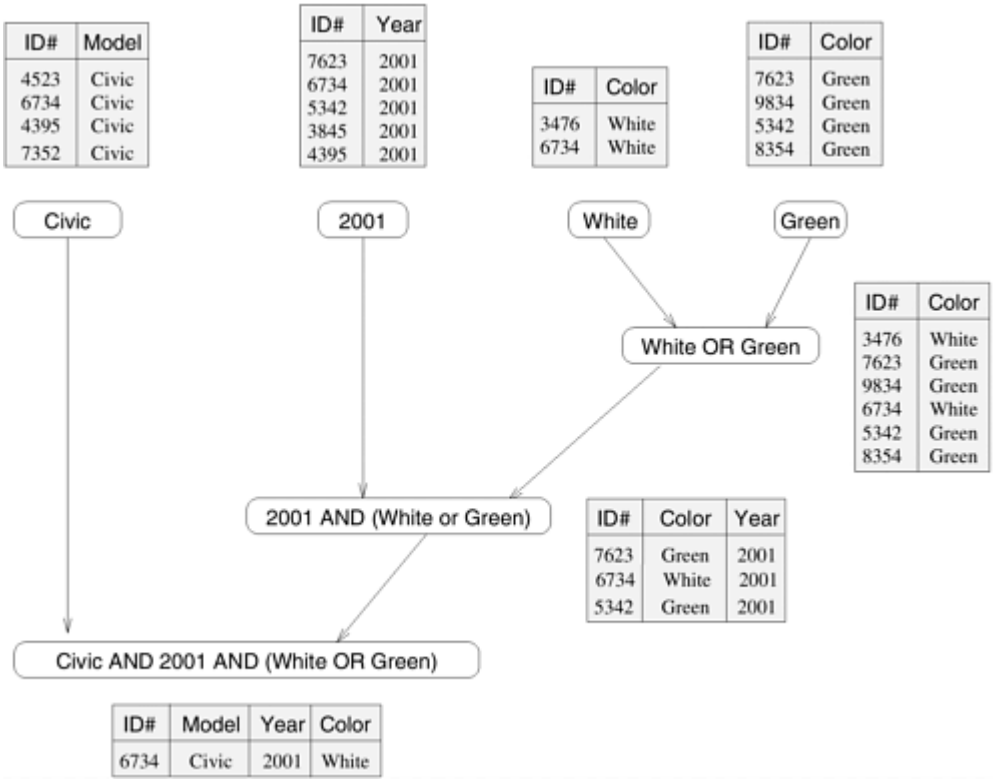
The various computations involved in processing the query in Example 3.2 can be visualized by the task-dependency graph shown in Figure 3.2. Each node in this figure is a task that corresponds to an intermediate table that needs to be computed and the arrows between nodes indicate dependencies between the tasks. For example, before we can compute the table that corresponds to the 2001 Civics, we must first compute the table of all the Civics and a table of all the 2001-model cars.

Figure 3.2. The different tables and their dependencies in a query processing operation.



Note that often there are multiple ways of expressing certain computations, especially those involving associative operators such as addition, multiplication, and logical AND or OR. Different ways of arranging computations can lead to different task-dependency graphs with different characteristics. For instance, the database query in Example 3.2 can be solved by first computing a table of all green or white cars, then performing an intersection with a table of all 2001 model cars, and finally combining the results with the table of all Civics. This sequence of computation results in the task-dependency graph shown in Figure 3.3.

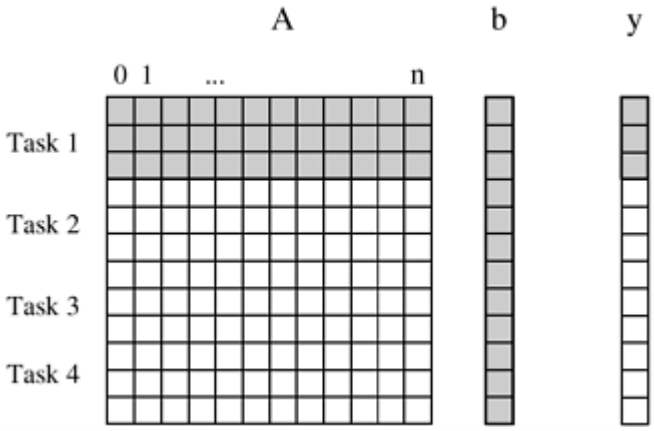
Figure 3.3. An alternate data-dependency graph for the query processing operation.



3.1.2 Granularity, Concurrency, and Task-Interaction

The number and size of tasks into which a problem is decomposed determines the *granularity* of the decomposition. A decomposition into a large number of small tasks is called *fine-grained* and a decomposition into a small number of large tasks is called *coarse-grained*. For example, the decomposition for matrix-vector multiplication shown in Figure 3.1 would usually be considered fine-grained because each of a large number of tasks performs a single dot-product. Figure 3.4 shows a coarse-grained decomposition of the same problem into four tasks, where each tasks computes $n/4$ of the entries of the output vector of length n.

Figure 3.4. Decomposition of dense matrix-vector multiplication into four tasks. The portions of the matrix and the input and output vectors accessed by Task 1 are highlighted.



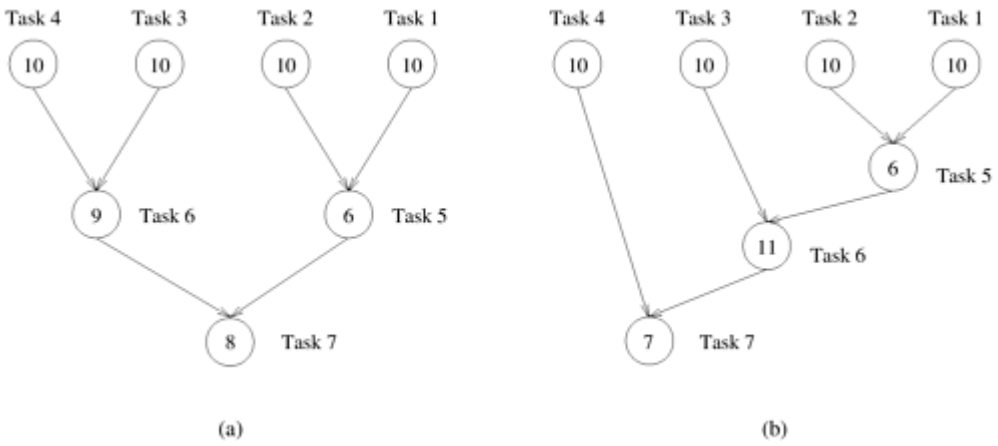
A concept related to granularity is that of *degree of concurrency*. The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its *maximum degree of concurrency*. In most cases, the maximum degree of concurrency is less than the total number of tasks due to dependencies among the tasks. For example, the maximum degree of concurrency in the task-graphs of Figures 3.2 and 3.3 is four. In these task-graphs, maximum concurrency is available right at the beginning when tables for Model, Year, Color Green, and Color White can be computed simultaneously. In general, for task-dependency graphs that are trees, the maximum degree of concurrency is always equal to the number of leaves in the tree.

A more useful indicator of a parallel program's performance is the *average degree of concurrency*, which is the average number of tasks that can run concurrently over the entire duration of execution of the program.

Both the maximum and the average degrees of concurrency usually increase as the granularity of tasks becomes smaller (finer). For example, the decomposition of matrix-vector multiplication shown in Figure 3.1 has a fairly small granularity and a large degree of concurrency. The decomposition for the same problem shown in Figure 3.4 has a larger granularity and a smaller degree of concurrency.

The degree of concurrency also depends on the shape of the task-dependency graph and the same granularity, in general, does not guarantee the same degree of concurrency. For example, consider the two task graphs in Figure 3.5, which are abstractions of the task graphs of Figures 3.2 and 3.3, respectively (Problem 3.1). The number inside each node represents the amount of work required to complete the task corresponding to that node. The average degree of concurrency of the task graph in Figure 3.5(a) is 2.33 and that of the task graph in Figure 3.5(b) is 1.88 (Problem 3.1), although both task-dependency graphs are based on the same decomposition.

Figure 3.5. Abstractions of the task graphs of Figures 3.2 and 3.3, respectively.



A feature of a task-dependency graph that determines the average degree of concurrency for a given granularity is its *critical path*. In a task-dependency graph, let us refer to the nodes with no incoming edges by *start nodes* and the nodes with no outgoing edges by *finish nodes*. The longest directed path between any pair of start and finish nodes is known as the critical path. The sum of the weights of nodes along this path is known as the *critical path length*, where the weight of a node is the size or the amount of work associated with the corresponding task. The ratio of the total amount of work to the critical-path length is the average degree of concurrency. Therefore, a shorter critical path favors a higher degree of concurrency. For example, the critical path length is 27 in the task-dependency graph shown in Figure 3.5(a) and

is 34 in the task-dependency graph shown in Figure 3.5(b). Since the total amount of work required to solve the problems using the two decompositions is 63 and 64, respectively, the average degree of concurrency of the two task-dependency graphs is 2.33 and 1.88, respectively.

Although it may appear that the time required to solve a problem can be reduced simply by increasing the granularity of decomposition and utilizing the resulting concurrency to perform more and more tasks in parallel, this is not the case in most practical scenarios. Usually, there is an inherent bound on how fine-grained a decomposition a problem permits. For instance, there are n^2 multiplications and additions in matrix-vector multiplication considered in Example 3.1 and the problem cannot be decomposed into more than $O(n^2)$ tasks even by using the most fine-grained decomposition.

Other than limited granularity and degree of concurrency, there is another important practical factor that limits our ability to obtain unbounded speedup (ratio of serial to parallel execution time) from parallelization. This factor is the *interaction* among tasks running on different physical processors. The tasks that a problem is decomposed into often share input, output, or intermediate data. The dependencies in a task-dependency graph usually result from the fact that the output of one task is the input for another. For example, in the database query example, tasks share intermediate data; the table generated by one task is often used by another task as input. Depending on the definition of the tasks and the parallel programming paradigm, there may be interactions among tasks that appear to be independent in a task-dependency graph. For example, in the decomposition for matrix-vector multiplication, although all tasks are independent, they all need access to the entire input vector *b*. Since originally there is only one copy of the vector *b*, tasks may have to send and receive messages for all of them to access the entire vector in the distributed-memory paradigm.

The pattern of interaction among tasks is captured by what is known as a *task-interaction graph*. The nodes in a task-interaction graph represent tasks and the edges connect tasks that interact with each other. The nodes and edges of a task-interaction graph can be assigned weights proportional to the amount of computation a task performs and the amount of interaction that occurs along an edge, if this information is known. The edges in a task-interaction graph are usually undirected, but directed edges can be used to indicate the direction of flow of data, if it is unidirectional. The edge-set of a task-interaction graph is usually a superset of the edge-set of the task-dependency graph. In the database query example discussed earlier, the task-interaction graph is the same as the task-dependency graph. We now give an example of a more interesting task-interaction graph that results from the problem of sparse matrix-vector multiplication.

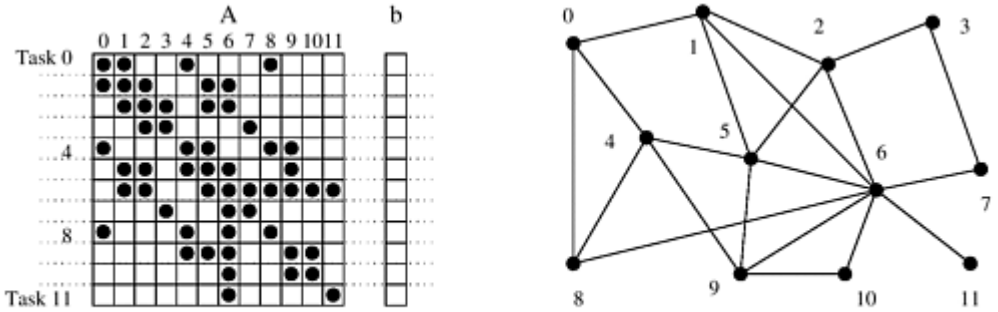
Example 3.3 Sparse matrix-vector multiplication

Consider the problem of computing the product $y = Ab$ of a sparse n x n matrix A with a dense n x 1 vector b. A matrix is considered sparse when a significant number of entries in it are zero and the locations of the non-zero entries do not conform to a predefined structure or pattern. Arithmetic operations involving sparse matrices can often be optimized significantly by avoiding computations involving the zeros. For instance, while computing the ith entry $y[i] = \sum_{j=1}^{n} (A[i, j] \times b[j])$ of the product vector, we need to compute the products $A[i, j] \times b[j]$ for only those values of j for which $A[i, j] \neq 0$. For example, $y[0] = A[0, 0].b[0] + A[0, 1].b[1] + A[0, 4].b[4] + A[0, 8].b[8]$.

One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it. Figure 3.6(a) illustrates this

decomposition. In addition to assigning the computation of the element $y[i]$ of the output vector to Task i, we also make it the "owner" of row $A[i, *]$ of the matrix and the element $b[i]$ of the input vector. Note that the computation of $y[i]$ requires access to many elements of b that are owned by other tasks. So Task i must get these elements from the appropriate locations. In the message-passing paradigm, with the ownership of $b[i]$,Task i also inherits the responsibility of sending $b[i]$ to all the other tasks that need it for their computation. For example, Task 4 must send $b[4]$ to Tasks 0, 5, 8, and 9 and must get $b[0]$, $b[5]$, $b[8]$, and $b[9]$ to perform its own computation. The resulting task-interaction graph is shown in Figure 3.6(b). ■

Figure 3.6. A decomposition for sparse matrix-vector multiplication and the corresponding task-interaction graph. In the decomposition Task i computes $\sum_{0 \leq j \leq 11, A[i,j] \neq 0} A[i, j].b[j]$.



Chapter 5 contains detailed quantitative analysis of overheads due to interaction and limited concurrency and their effect on the performance and scalability of parallel algorithm-architecture combinations. In this section, we have provided a basic introduction to these factors because they require important consideration in designing parallel algorithms.

3.1.3 Processes and Mapping

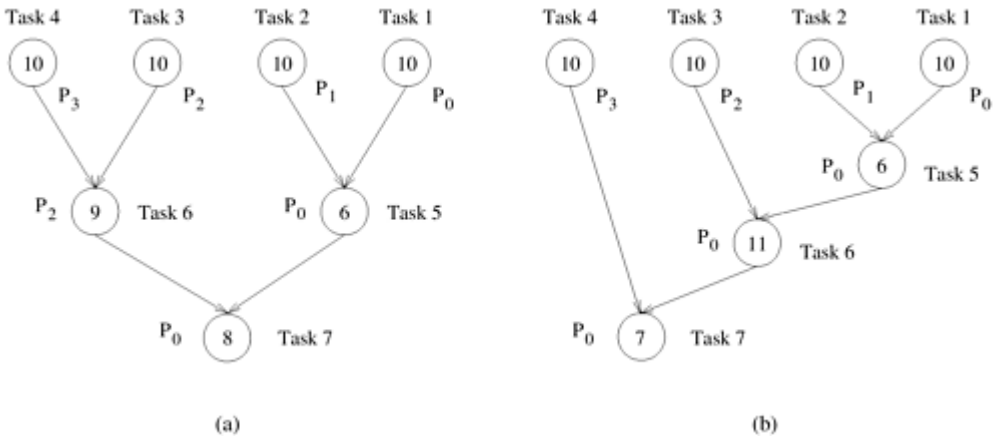
The tasks, into which a problem is decomposed, run on physical processors. However, for reasons that we shall soon discuss, we will use the term *process* in this chapter to refer to a processing or computing agent that performs tasks. In this context, the term process does not adhere to the rigorous operating system definition of a process. Instead, it is an abstract entity that uses the code and data corresponding to a task to produce the output of that task within a finite amount of time after the task is activated by the parallel program. During this time, in addition to performing computations, a process may synchronize or communicate with other processes, if needed. In order to obtain any speedup over a sequential implementation, a parallel program must have several processes active simultaneously, working on different tasks. The mechanism by which tasks are assigned to processes for execution is called *mapping*. For example, four processes could be assigned the task of computing one submatrix of C each in the matrix-multiplication computation of Example 3.5.

The task-dependency and task-interaction graphs that result from a choice of decomposition play an important role in the selection of a good mapping for a parallel algorithm. A good mapping should seek to maximize the use of concurrency by mapping independent tasks onto different processes, it should seek to minimize the total completion time by ensuring that processes are available to execute the tasks on the critical path as soon as such tasks become

executable, and it should seek to minimize interaction among processes by mapping tasks with a high degree of mutual interaction onto the same process. In most nontrivial parallel algorithms, these tend to be conflicting goals. For instance, the most efficient decomposition-mapping combination is a single task mapped onto a single process. It wastes no time in idling or interacting, but achieves no speedup either. Finding a balance that optimizes the overall parallel performance is the key to a successful parallel algorithm. Therefore, mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel algorithm is. Even though the degree of concurrency is determined by the decomposition, it is the mapping that determines how much of that concurrency is actually utilized, and how efficiently.

For example, Figure 3.7 shows efficient mappings for the decompositions and the task-interaction graphs of Figure 3.5 onto four processes. Note that, in this case, a maximum of four processes can be employed usefully, although the total number of tasks is seven. This is because the maximum degree of concurrency is only four. The last three tasks can be mapped arbitrarily among the processes to satisfy the constraints of the task-dependency graph. However, it makes more sense to map the tasks connected by an edge onto the same process because this prevents an inter-task interaction from becoming an inter-processes interaction. For example, in Figure 3.7(b), if Task 5 is mapped onto process P_2, then both processes P_0 and P_1 will need to interact with P_2. In the current mapping, only a single interaction between P_0 and P_1 suffices.

Figure 3.7. Mappings of the task graphs of Figure 3.5 onto four processes.



3.1.4 Processes versus Processors

In the context of parallel algorithm design, processes are logical computing agents that perform tasks. Processors are the hardware units that physically perform computations. In this text, we choose to express parallel algorithms and programs in terms of processes. In most cases, when we refer to processes in the context of a parallel algorithm, there is a one-to-one correspondence between processes and processors and it is appropriate to assume that there are as many processes as the number of physical CPUs on the parallel computer. However, sometimes a higher level of abstraction may be required to express a parallel algorithm, especially if it is a complex algorithm with multiple stages or with different forms of parallelism.

Treating processes and processors separately is also useful when designing parallel programs for hardware that supports multiple programming paradigms. For instance, consider a parallel computer that consists of multiple computing nodes that communicate with each other via message passing. Now each of these nodes could be a shared-address-space module with

multiple CPUs. Consider implementing matrix multiplication on such a parallel computer. The best way to design a parallel algorithm is to do so in two stages. First, develop a decomposition and mapping strategy suitable for the message-passing paradigm and use this to exploit parallelism among the nodes. Each task that the original matrix multiplication problem decomposes into is a matrix multiplication computation itself. The next step is to develop a decomposition and mapping strategy suitable for the shared-memory paradigm and use this to implement each task on the multiple CPUs of a node.

PREVIOUS NEXT

3.2 Decomposition Techniques

As mentioned earlier, one of the fundamental steps that we need to undertake to solve a problem in parallel is to split the computations to be performed into a set of tasks for concurrent execution defined by the task-dependency graph. In this section, we describe some commonly used decomposition techniques for achieving concurrency. This is not an exhaustive set of possible decomposition techniques. Also, a given decomposition is not always guaranteed to lead to the best parallel algorithm for a given problem. Despite these shortcomings, the decomposition techniques described in this section often provide a good starting point for many problems and one or a combination of these techniques can be used to obtain effective decompositions for a large variety of problems.

These techniques are broadly classified as *recursive decomposition*, *data-decomposition*, *exploratory decomposition*, and *speculative decomposition*. The recursive- and data-decomposition techniques are relatively *general purpose* as they can be used to decompose a wide variety of problems. On the other hand, speculative- and exploratory-decomposition techniques are more of a *special purpose* nature because they apply to specific classes of problems.

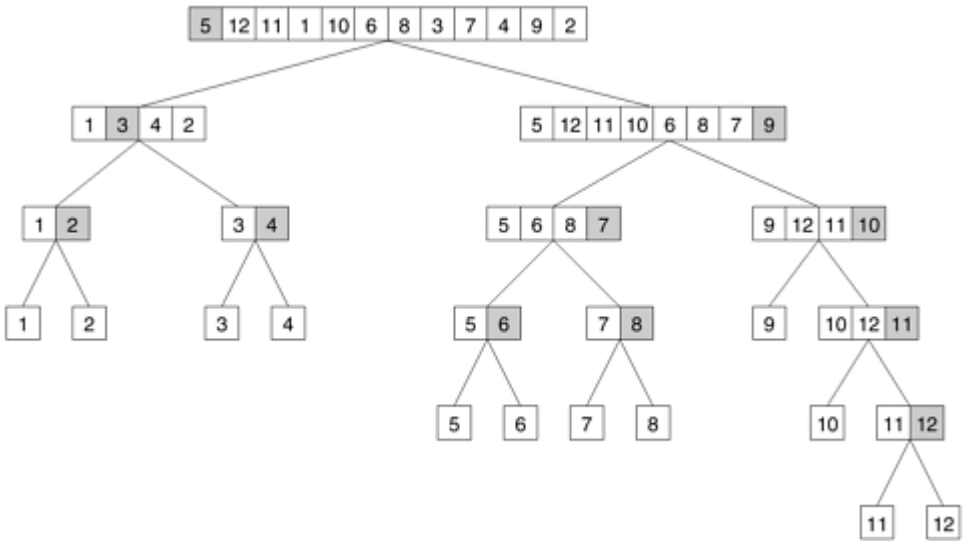
3.2.1 Recursive Decomposition

Recursive decomposition is a method for inducing concurrency in problems that can be solved using the divide-and-conquer strategy. In this technique, a problem is solved by first dividing it into a set of independent subproblems. Each one of these subproblems is solved by recursively applying a similar division into smaller subproblems followed by a combination of their results. The divide-and-conquer strategy results in natural concurrency, as different subproblems can be solved concurrently.

Example 3.4 Quicksort

Consider the problem of sorting a sequence A of n elements using the commonly used quicksort algorithm. Quicksort is a divide and conquer algorithm that starts by selecting a pivot element x and then partitions the sequence A into two subsequences A_0 and A_1 such that all the elements in A_0 are smaller than x and all the elements in A_1 are greater than or equal to x. This partitioning step forms the *divide* step of the algorithm. Each one of the subsequences A_0 and A_1 is sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. This is illustrated in Figure 3.8 for a sequence of 12 numbers. The recursion terminates when each subsequence contains only a single element. ■

Figure 3.8. The quicksort task-dependency graph based on recursive decomposition for sorting a sequence of 12 numbers.



In Figure 3.8, we define a task as the work of partitioning a given subsequence. Therefore, Figure 3.8 also represents the task graph for the problem. Initially, there is only one sequence (i.e., the root of the tree), and we can use only a single process to partition it. The completion of the root task results in two subsequences (A_0 and A_1, corresponding to the two nodes at the first level of the tree) and each one can be partitioned in parallel. Similarly, the concurrency continues to increase as we move down the tree.

Sometimes, it is possible to restructure a computation to make it amenable to recursive decomposition even if the commonly used algorithm for the problem is not based on the divide-and-conquer strategy. For example, consider the problem of finding the minimum element in an unordered sequence A of n elements. The serial algorithm for solving this problem scans the entire sequence A, recording at each step the minimum element found so far as illustrated in Algorithm 3.1. It is easy to see that this serial algorithm exhibits no concurrency.

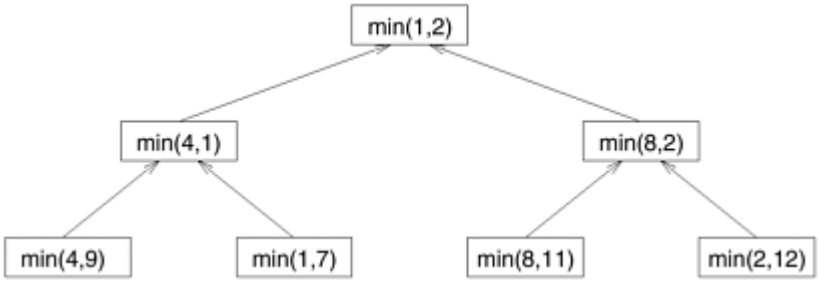
Algorithm 3.1 A serial program for finding the minimum in an array of numbers A of length n.

```
1.    procedure SERIAL_MIN (A, n)
2.    begin
3.    min = A[0];
4.    for i := 1 to n - 1 do
5.        if (A[i] < min) min := A[i];
6.    endfor;
7.    return min;
8.    end SERIAL_MIN
```

Once we restructure this computation as a divide-and-conquer algorithm, we can use recursive decomposition to extract concurrency. Algorithm 3.2 is a divide-and-conquer algorithm for finding the minimum element in an array. In this algorithm, we split the sequence A into two subsequences, each of size $n/2$, and we find the minimum for each of these subsequences by performing a recursive call. Now the overall minimum element is found by selecting the minimum of these two subsequences. The recursion terminates when there is only one element left in each subsequence. Having restructured the serial computation in this manner, it is easy to construct a task-dependency graph for this problem. Figure 3.9 illustrates such a task-dependency graph for finding the minimum of eight numbers where each task is assigned the

work of finding the minimum of two numbers.

Figure 3.9. The task-dependency graph for finding the minimum number in the sequence {4, 9, 1, 7, 8, 11, 2, 12}. Each node in the tree represents the task of finding the minimum of a pair of numbers.



Algorithm 3.2 A recursive program for finding the minimum in an array of numbers *A* of length *n*.

```
1.   procedure RECURSIVE_MIN (A, n)
2.   begin
3.   if (n = 1) then
4.       min := A[0];
5.   else
6.       lmin := RECURSIVE_MIN (A, n/2);
7.       rmin := RECURSIVE_MIN (&(A[n/2]), n - n/2);
8.       if (lmin < rmin) then
9.           min := lmin;
10.      else
11.          min := rmin;
12.      endelse;
13.  endelse;
14.  return min;
15.  end RECURSIVE_MIN
```

3.2.2 Data Decomposition

Data decomposition is a powerful and commonly used method for deriving concurrency in algorithms that operate on large data structures. In this method, the decomposition of computations is done in two steps. In the first step, the data on which the computations are performed is partitioned, and in the second step, this data partitioning is used to induce a partitioning of the computations into tasks. The operations that these tasks perform on different data partitions are usually similar (e.g., matrix multiplication introduced in Example 3.5) or are chosen from a small set of operations (e.g., LU factorization introduced in Example 3.10).

The partitioning of data can be performed in many possible ways as discussed next. In general, one must explore and evaluate all possible ways of partitioning the data and determine which one yields a natural and efficient computational decomposition.

Partitioning Output Data In many computations, each element of the output can be computed independently of others as a function of the input. In such computations, a partitioning of the output data automatically induces a decomposition of the problems into tasks, where each task is assigned the work of computing a portion of the output. We introduce the problem of matrix-

multiplication in Example 3.5 to illustrate a decomposition based on partitioning output data.

Example 3.5 Matrix multiplication

Consider the problem of multiplying two $n \times n$ matrices A and B to yield a matrix C. Figure 3.10 shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting each dimension of the matrix into half. The four submatrices of C, roughly of size $n/2 \times n/2$ each, are then independently computed by four tasks as the sums of the appropriate products of submatrices of A and B. ■

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)

$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

(b)

Most matrix algorithms, including matrix-vector and matrix-matrix multiplication, can be formulated in terms of block matrix operations. In such a formulation, the matrix is viewed as composed of blocks or submatrices and the scalar arithmetic operations on its elements are replaced by the equivalent matrix operations on the blocks. The results of the element and the block versions of the algorithm are mathematically equivalent (Problem 3.10). Block versions of matrix algorithms are often used to aid decomposition.

The decomposition shown in Figure 3.10 is based on partitioning the output matrix C into four submatrices and each of the four tasks computes one of these submatrices. The reader must note that data-decomposition is distinct from the decomposition of the computation into tasks. Although the two are often related and the former often aids the latter, a given data-decomposition does not result in a unique decomposition into tasks. For example, Figure 3.11 shows two other decompositions of matrix multiplication, each into eight tasks, corresponding to the same data-decomposition as used in Figure 3.10(a).

Figure 3.11. Two examples of decomposition of matrix multiplication into eight tasks.

Decomposition I	Decomposition II
Task 1: $C_{1,1} = A_{1,1}B_{1,1}$	Task 1: $C_{1,1} = A_{1,1}B_{1,1}$
Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$	Task 2: $C_{1,1} = C_{1,1} + A_{1,2}B_{2,1}$
Task 3: $C_{1,2} = A_{1,1}B_{1,2}$	Task 3: $C_{1,2} = A_{1,2}B_{2,2}$
Task 4: $C_{1,2} = C_{1,2} + A_{1,2}B_{2,2}$	Task 4: $C_{1,2} = C_{1,2} + A_{1,1}B_{1,2}$
Task 5: $C_{2,1} = A_{2,1}B_{1,1}$	Task 5: $C_{2,1} = A_{2,2}B_{2,1}$
Task 6: $C_{2,1} = C_{2,1} + A_{2,2}B_{2,1}$	Task 6: $C_{2,1} = C_{2,1} + A_{2,1}B_{1,1}$
Task 7: $C_{2,2} = A_{2,1}B_{1,2}$	Task 7: $C_{2,2} = A_{2,1}B_{1,2}$
Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$	Task 8: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

We now introduce another example to illustrate decompositions based on data partitioning. Example 3.6 describes the problem of computing the frequency of a set of itemsets in a transaction database, which can be decomposed based on the partitioning of output data.

Example 3.6 Computing frequencies of itemsets in a transaction database

Consider the problem of computing the frequency of a set of itemsets in a transaction database. In this problem we are given a set T containing n transactions and a set I containing m itemsets. Each transaction and itemset contains a small number of items, out of a possible set of items. For example, T could be a grocery stores database of customer sales with each transaction being an individual grocery list of a shopper and each itemset could be a group of items in the store. If the store desires to find out how many customers bought each of the designated groups of items, then it would need to find the number of times that each itemset in I appears in all the transactions; i.e., the number of transactions of which each itemset is a subset of. Figure 3.12(a) shows an example of this type of computation. The database shown in Figure 3.12 consists of 10 transactions, and we are interested in computing the frequency of the eight itemsets shown in the second column. The actual frequencies of these itemsets in the database, which are the output of the frequency-computing program, are shown in the third column. For instance, itemset {D, K} appears twice, once in the second and once in the ninth transaction. ■

Figure 3.12. Computing itemset frequencies in a transaction database.

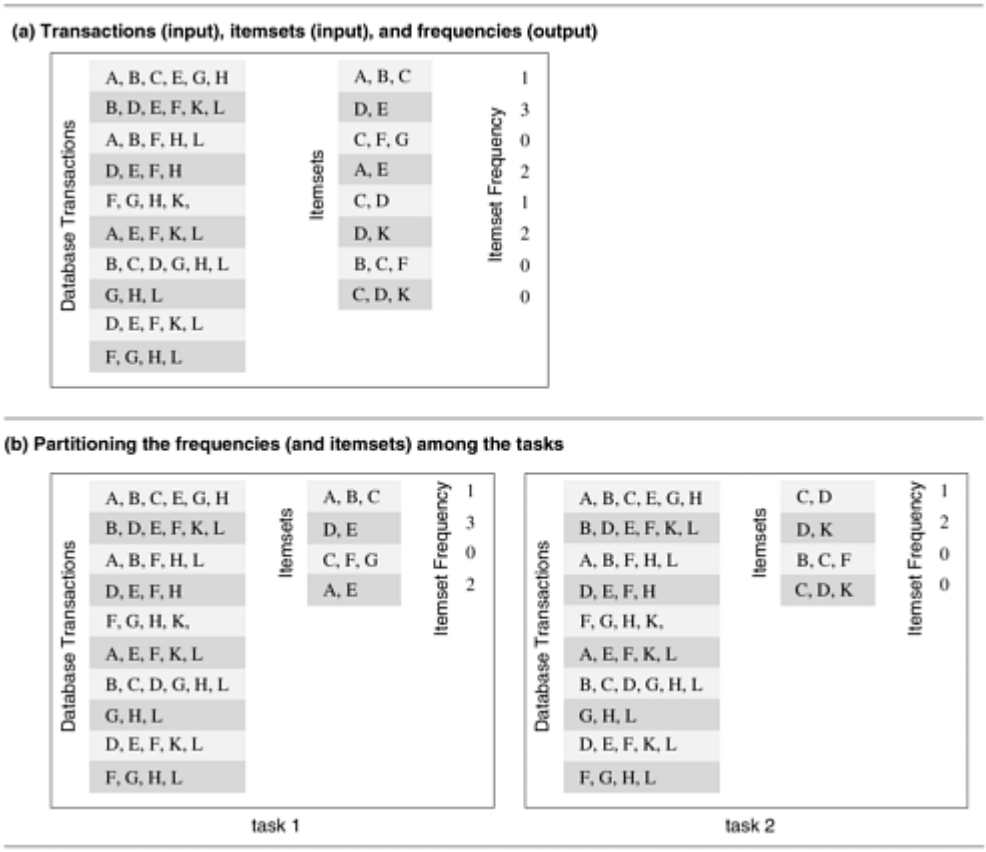
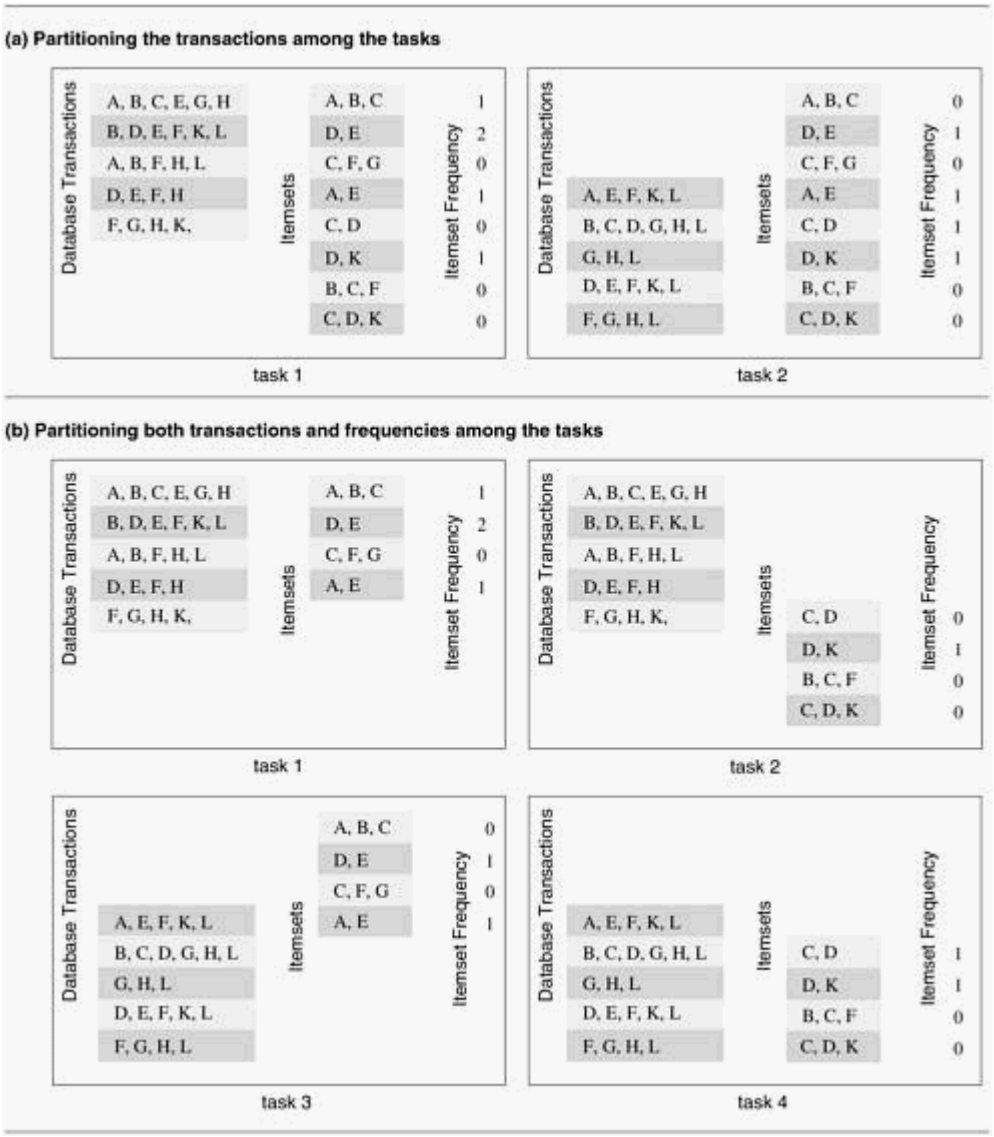


Figure 3.12(b) shows how the computation of frequencies of the itemsets can be decomposed into two tasks by partitioning the output into two parts and having each task compute its half of the frequencies. Note that, in the process, the itemsets input has also been partitioned, but the primary motivation for the decomposition of Figure 3.12(b) is to have each task independently compute the subset of frequencies assigned to it.

Partitioning Input Data Partitioning of output data can be performed only if each output can be naturally computed as a function of the input. In many algorithms, it is not possible or desirable to partition the output data. For example, while finding the minimum, maximum, or the sum of a set of numbers, the output is a single unknown value. In a sorting algorithm, the individual elements of the output cannot be efficiently determined in isolation. In such cases, it is sometimes possible to partition the input data, and then use this partitioning to induce concurrency. A task is created for each partition of the input data and this task performs as much computation as possible using these local data. Note that the solutions to tasks induced by input partitions may not directly solve the original problem. In such cases, a follow-up computation is needed to combine the results. For example, while finding the sum of a sequence of N numbers using p processes ($N > p$), we can partition the input into p subsets of nearly equal sizes. Each task then computes the sum of the numbers in one of the subsets. Finally, the p partial results can be added up to yield the final result.

The problem of computing the frequency of a set of itemsets in a transaction database described in Example 3.6 can also be decomposed based on a partitioning of input data. Figure 3.13(a) shows a decomposition based on a partitioning of the input set of transactions. Each of the two tasks computes the frequencies of all the itemsets in its respective subset of transactions. The two sets of frequencies, which are the independent outputs of the two tasks, represent intermediate results. Combining the intermediate results by pairwise addition yields the final result.

Figure 3.13. Some decompositions for computing itemset frequencies in a transaction database.

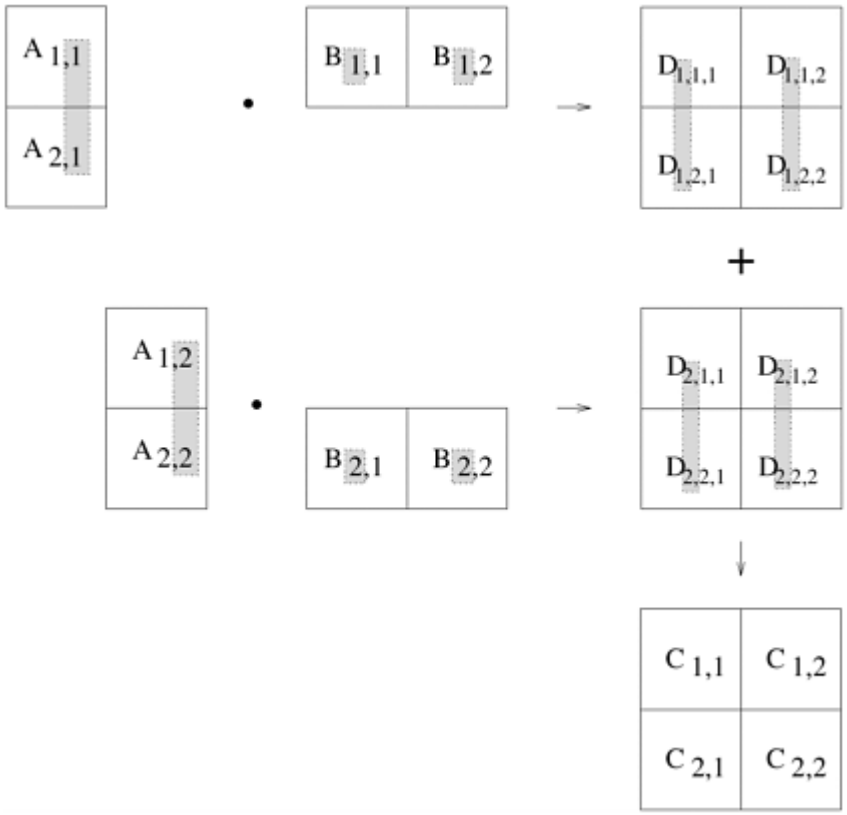


Partitioning both Input and Output Data In some cases, in which it is possible to partition the output data, partitioning of input data can offer additional concurrency. For example, consider the 4-way decomposition shown in Figure 3.13(b) for computing itemset frequencies. Here, both the transaction set and the frequencies are divided into two parts and a different one of the four possible combinations is assigned to each of the four tasks. Each task then computes a local set of frequencies. Finally, the outputs of Tasks 1 and 3 are added together, as are the outputs of Tasks 2 and 4.

Partitioning Intermediate Data Algorithms are often structured as multi-stage computations such that the output of one stage is the input to the subsequent stage. A decomposition of such an algorithm can be derived by partitioning the input or the output data of an intermediate stage of the algorithm. Partitioning intermediate data can sometimes lead to higher concurrency than partitioning input or output data. Often, the intermediate data are not generated explicitly in the serial algorithm for solving the problem and some restructuring of the original algorithm may be required to use intermediate data partitioning to induce a decomposition.

Let us revisit matrix multiplication to illustrate a decomposition based on partitioning intermediate data. Recall that the decompositions induced by a 2 x 2 partitioning of the output matrix C, as shown in Figures 3.10 and 3.11, have a maximum degree of concurrency of four. We can increase the degree of concurrency by introducing an intermediate stage in which eight tasks compute their respective product submatrices and store the results in a temporary three-dimensional matrix D, as shown in Figure 3.14. The submatrix $D_{k,i,j}$ is the product of $A_{i,k}$ and $B_{k,j}$.

Figure 3.14. Multiplication of matrices A and B with partitioning of the three-dimensional intermediate matrix D.



A partitioning of the intermediate matrix D induces a decomposition into eight tasks. Figure 3.15 shows this decomposition. After the multiplication phase, a relatively inexpensive matrix addition step can compute the result matrix C. All submatrices $D_{*,i,j}$ with the same second and third dimensions i and j are added to yield $C_{i,j}$. The eight tasks numbered 1 through 8 in Figure 3.15 perform $O(n^3/8)$ work each in multiplying $n/2$ x $n/2$ submatrices of A and B. Then, four tasks numbered 9 through 12 spend $O(n^2/4)$ time each in adding the appropriate $n/2$ x $n/2$ submatrices of the intermediate matrix D to yield the final result matrix C. Figure 3.16 shows the task-dependency graph corresponding to the decomposition shown in Figure 3.15.

Figure 3.15. A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

Stage I

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

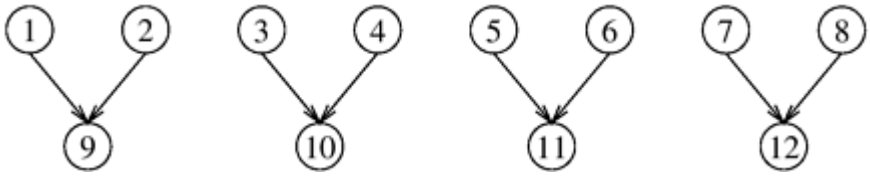
Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

A decomposition induced by a partitioning of D

$$\begin{aligned}
\text{Task 01:} \quad & D_{1,1,1} = A_{1,1}B_{1,1} \\
\text{Task 02:} \quad & D_{2,1,1} = A_{1,2}B_{2,1} \\
\text{Task 03:} \quad & D_{1,1,2} = A_{1,1}B_{1,2} \\
\text{Task 04:} \quad & D_{2,1,2} = A_{1,2}B_{2,2} \\
\text{Task 05:} \quad & D_{1,2,1} = A_{2,1}B_{1,1} \\
\text{Task 06:} \quad & D_{2,2,1} = A_{2,2}B_{2,1} \\
\text{Task 07:} \quad & D_{1,2,2} = A_{2,1}B_{1,2} \\
\text{Task 08:} \quad & D_{2,2,2} = A_{2,2}B_{2,2} \\
\text{Task 09:} \quad & C_{1,1} = D_{1,1,1} + D_{2,1,1} \\
\text{Task 10:} \quad & C_{1,2} = D_{1,1,2} + D_{2,1,2} \\
\text{Task 11:} \quad & C_{2,1} = D_{1,2,1} + D_{2,2,1} \\
\text{Task 12:} \quad & C_{2,2} = D_{1,2,2} + D_{2,2,2}
\end{aligned}$$

Figure 3.16. The task-dependency graph of the decomposition shown in Figure 3.15.



Note that all elements of D are computed implicitly in the original decomposition shown in Figure 3.11, but are not explicitly stored. By restructuring the original algorithm and by explicitly storing D, we have been able to devise a decomposition with higher concurrency. This, however, has been achieved at the cost of extra aggregate memory usage.

The Owner-Computes Rule A decomposition based on partitioning output or input data is also widely referred to as the *owner-computes* rule. The idea behind this rule is that each partition performs all the computations involving data that it owns. Depending on the nature of the data or the type of data-partitioning, the owner-computes rule may mean different things. For instance, when we assign partitions of the input data to tasks, then the owner-computes rule means that a task performs all the computations that can be done using these data. On the other hand, if we partition the output data, then the owner-computes rule means that a task computes all the data in the partition assigned to it.

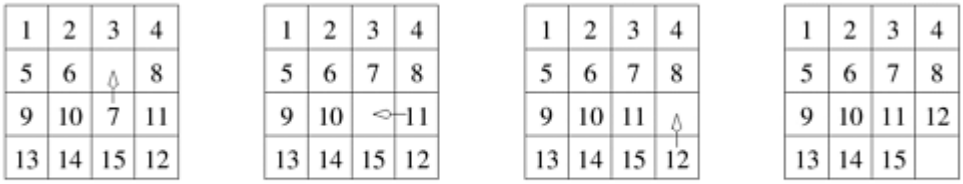
3.2.3 Exploratory Decomposition

Exploratory decomposition is used to decompose problems whose underlying computations correspond to a search of a space for solutions. In exploratory decomposition, we partition the search space into smaller parts, and search each one of these parts concurrently, until the desired solutions are found. For an example of exploratory decomposition, consider the 15-puzzle problem.

Example 3.7 The 15-puzzle problem

The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4 x 4 grid. A tile can be moved into the blank position from a position adjacent to it, thus creating a blank in the tile's original position. Depending on the configuration of the grid, up to four moves are possible: up, down, left, and right. The initial and final configurations of the tiles are specified. The objective is to determine any sequence or a shortest sequence of moves that transforms the initial configuration to the final configuration. Figure 3.17 illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration. ■

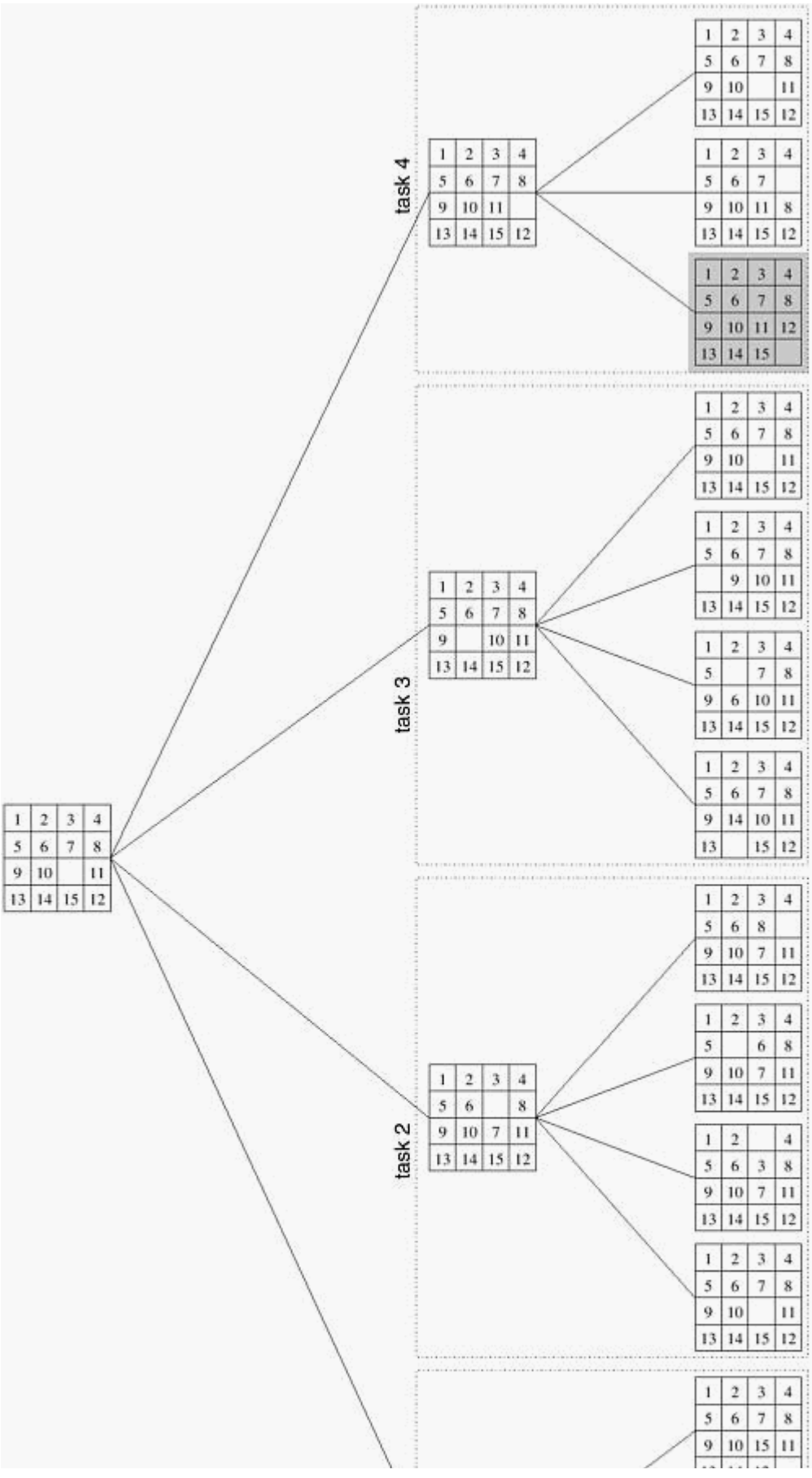
Figure 3.17. A 15-puzzle problem instance showing the initial configuration (a), the final configuration (d), and a sequence of moves leading from the initial to the final configuration.

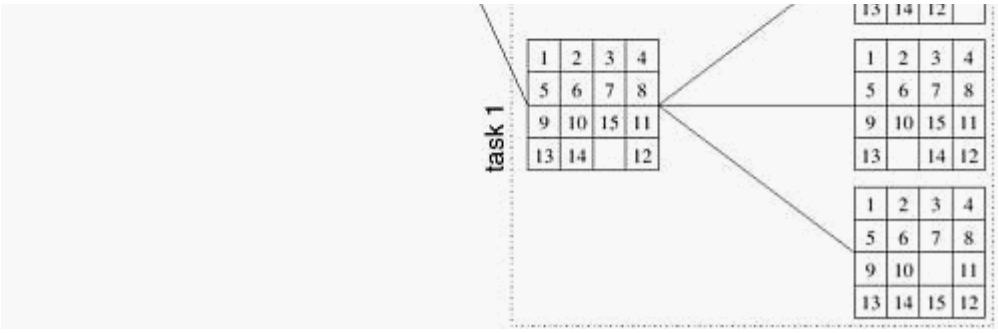


The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successor configurations are generated. A configuration may have 2, 3, or 4 possible successor configurations, each corresponding to the occupation of the empty slot by one of its neighbors. The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated configurations to the final configuration. Since one of these newly generated configurations must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution. The configuration space generated by the tree search is often referred to as a state space graph. Each node of the graph is a configuration and each edge of the graph connects configurations that can be reached from one another by a single move of a tile.

One method for solving this problem in parallel is as follows. First, a few levels of configurations starting from the initial configuration are generated serially until the search tree has a sufficient number of leaf nodes (i.e., configurations of the 15-puzzle). Now each node is assigned to a task to explore further until at least one of them finds a solution. As soon as one of the concurrent tasks finds a solution it can inform the others to terminate their searches. Figure 3.18 illustrates one such decomposition into four tasks in which task 4 finds the solution.

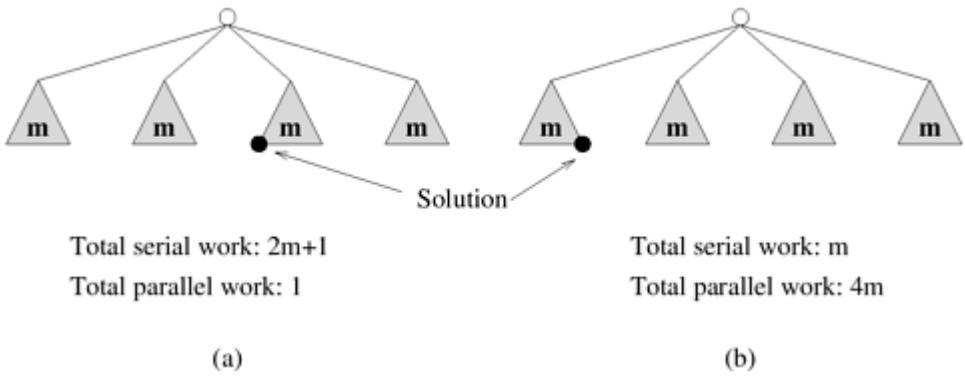
Figure 3.18. The states generated by an instance of the 15-puzzle problem.


task 4
task 3
task 2



Note that even though exploratory decomposition may appear similar to data-decomposition (the search space can be thought of as being the data that get partitioned) it is fundamentally different in the following way. The tasks induced by data-decomposition are performed in their entirety and each task performs useful computations towards the solution of the problem. On the other hand, in exploratory decomposition, unfinished tasks can be terminated as soon as an overall solution is found. Hence, the portion of the search space searched (and the aggregate amount of work performed) by a parallel formulation can be very different from that searched by a serial algorithm. The work performed by the parallel formulation can be either smaller or greater than that performed by the serial algorithm. For example, consider a search space that has been partitioned into four concurrent tasks as shown in Figure 3.19. If the solution lies right at the beginning of the search space corresponding to task 3 (Figure 3.19(a)), then it will be found almost immediately by the parallel formulation. The serial algorithm would have found the solution only after performing work equivalent to searching the entire space corresponding to tasks 1 and 2. On the other hand, if the solution lies towards the end of the search space corresponding to task 1 (Figure 3.19(b)), then the parallel formulation will perform almost four times the work of the serial algorithm and will yield no speedup.

Figure 3.19. An illustration of anomalous speedups resulting from exploratory decomposition.



3.2.4 Speculative Decomposition

Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of other computations that precede it. In this situation, while one task is performing the computation whose output is used in deciding the next computation, other tasks can concurrently start the computations of the next stage. This scenario is similar to evaluating one or more of the branches of a *switch* statement in C in parallel before the input for the *switch* is available. While one task is performing the computation that will eventually resolve the switch, other tasks could pick up the multiple branches of the switch in parallel. When the input for the *switch* has finally been

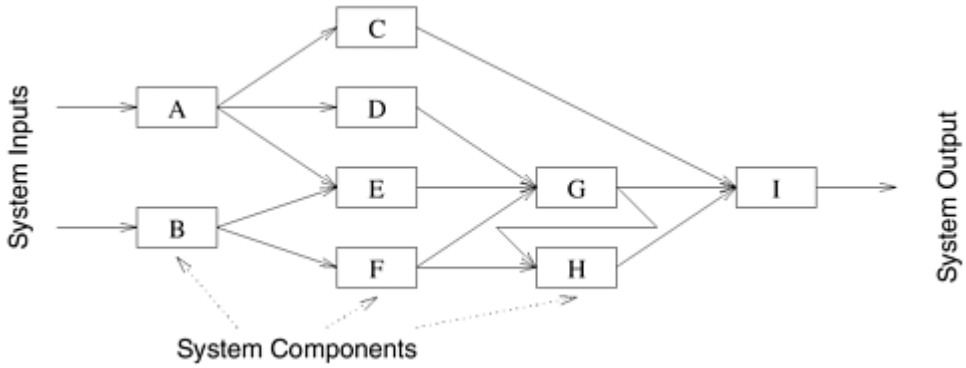
computed, the computation corresponding to the correct branch would be used while that corresponding to the other branches would be discarded. The parallel run time is smaller than the serial run time by the amount of time required to evaluate the condition on which the next task depends because this time is utilized to perform a useful computation for the next stage in parallel. However, this parallel formulation of a switch guarantees at least some wasteful computation. In order to minimize the wasted computation, a slightly different formulation of speculative decomposition could be used, especially in situations where one of the outcomes of the switch is more likely than the others. In this case, only the most promising branch is taken up a task in parallel with the preceding computation. In case the outcome of the switch is different from what was anticipated, the computation is rolled back and the correct branch of the switch is taken.

The speedup due to speculative decomposition can add up if there are multiple speculative stages. An example of an application in which speculative decomposition is useful is *discrete event simulation*. A detailed description of discrete event simulation is beyond the scope of this chapter; however, we give a simplified description of the problem.

Example 3.8 Parallel discrete event simulation

Consider the simulation of a system that is represented as a network or a directed graph. The nodes of this network represent components. Each component has an input buffer of jobs. The initial state of each component or node is idle. An idle component picks up a job from its input queue, if there is one, processes that job in some finite amount of time, and puts it in the input buffer of the components which are connected to it by outgoing edges. A component has to wait if the input buffer of one of its outgoing neighbors if full, until that neighbor picks up a job to create space in the buffer. There is a finite number of input job types. The output of a component (and hence the input to the components connected to it) and the time it takes to process a job is a function of the input job. The problem is to simulate the functioning of the network for a given sequence or a set of sequences of input jobs and compute the total completion time and possibly other aspects of system behavior. Figure 3.20 shows a simple network for a discrete event solution problem. ■

Figure 3.20. A simple network for discrete event simulation.



The problem of simulating a sequence of input jobs on the network described in Example 3.8 appears inherently sequential because the input of a typical component is the output of another. However, we can define speculative tasks that start simulating a subpart of the network, each

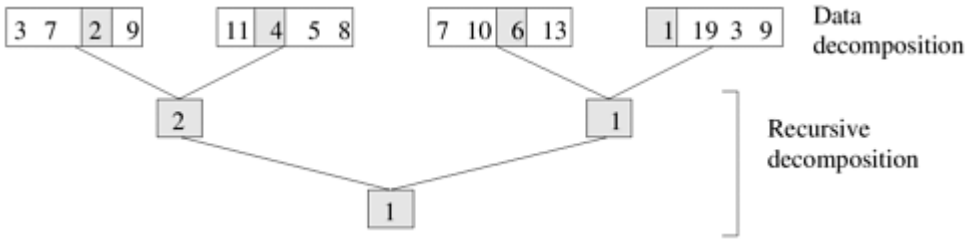
assuming one of several possible inputs to that stage. When an actual input to a certain stage becomes available (as a result of the completion of another selector task from a previous stage), then all or part of the work required to simulate this input would have already been finished if the speculation was correct, or the simulation of this stage is restarted with the most recent correct input if the speculation was incorrect.

Speculative decomposition is different from exploratory decomposition in the following way. In speculative decomposition, the input at a branch leading to multiple parallel tasks is unknown, whereas in exploratory decomposition, the output of the multiple tasks originating at a branch is unknown. In speculative decomposition, the serial algorithm would strictly perform only one of the tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take. Therefore, by preemptively computing for multiple possibilities out of which only one materializes, a parallel program employing speculative decomposition performs more aggregate work than its serial counterpart. Even if only one of the possibilities is explored speculatively, the parallel algorithm may perform more or the same amount of work as the serial algorithm. On the other hand, in exploratory decomposition, the serial algorithm too may explore different alternatives one after the other, because the branch that may lead to the solution is not known beforehand. Therefore, the parallel program may perform more, less, or the same amount of aggregate work compared to the serial algorithm depending on the location of the solution in the search space.

3.2.5 Hybrid Decompositions

So far we have discussed a number of decomposition methods that can be used to derive concurrent formulations of many algorithms. These decomposition techniques are not exclusive, and can often be combined together. Often, a computation is structured into multiple stages and it is sometimes necessary to apply different types of decomposition in different stages. For example, while finding the minimum of a large set of n numbers, a purely recursive decomposition may result in far more tasks than the number of processes, P, available. An efficient decomposition would partition the input into P roughly equal parts and have each task compute the minimum of the sequence assigned to it. The final result can be obtained by finding the minimum of the P intermediate results by using the recursive decomposition shown in Figure 3.21.

Figure 3.21. Hybrid decomposition for finding the minimum of an array of size 16 using four tasks.



As another example of an application of hybrid decomposition, consider performing quicksort in parallel. In Example 3.4, we used a recursive decomposition to derive a concurrent formulation of quicksort. This formulation results in $O(n)$ tasks for the problem of sorting a sequence of size n. But due to the dependencies among these tasks and due to uneven sizes of the tasks, the effective concurrency is quite limited. For example, the first task for splitting the input list into two parts takes $O(n)$ time, which puts an upper limit on the performance gain possible via parallelization. But the step of splitting lists performed by tasks in parallel quicksort can also be decomposed using the input decomposition technique discussed in Section 9.4.1. The resulting hybrid decomposition that combines recursive decomposition and the input data-decomposition

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 a 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.

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 Section 3.1.1. In the decomposition shown in Figure 3.10, all tasks are identical; they are applied to different data.

3.6.2 The Task Graph Model

As discussed in Section 3.1, 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 (Problem 3.5). 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 (Section 9.4.1), 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*.

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 (Section 11.4.4) 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.

Parallelization of loops by chunk scheduling (Section 3.4.2) 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.

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.

An example of a two-dimensional pipeline is the parallel LU factorization algorithm, which is discussed in detail in Section 8.3.1.

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 supertask comprising multiple subtasks that may be suitable for data-parallel or pipelined parallelism. Parallel quicksort (Sections 3.2.5 and 9.4.1) is one of the applications for which a hybrid model is ideally suited.