Contents

CHAPTER 4

Principles of Parallel Algorithm Design 1

CHAPTER 4

Principles of Parallel Algorithm Design

Algorithm development is critical to using computers to solve problems. A sequential algorithm is nothing more than 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 entails 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 in parallel. This is essential for obtaining any speed benefit from the use of a parallel computer. In practice, the specification of a non-trivial parallel program includes identifying portions of the work that can be performed concurrently, mapping the concurrent pieces of work onto multiple processors, distributing the input, output, and intermediate data associated with the program, managing accesses to data shared by multiple processors, and synchronizing the processors at various stages of the parallel program execution, if required. Typically, there are several choices for each of these, but usually, relatively few choices lead to a parallel algorithm that yields performance commensurate with the computational and storage resources employed to solve the problem. Often, a different choice may yield the best performance 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 the parallel program lies upon the parallel programmer or the algorithm designer. Tools and compilers for automatic parallelization at the current state of the art have limited applicability and success rate. Therefore, we do not consider these in this chapter or elsewhere in this book.

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

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

Figure 4.1 A decomposition of matrix multiplication into four tasks.

4.1 Introduction to Parallel Algorithm Design

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 multiplication, database query processing, LU factorization, and sparse matrix-vector multiplication as examples.

4.1.1 Tasks and Decomposition

Tasks are programmer-defined smaller units of computation into which the main computation is subdivided so that at least some of them can execute simultaneously to reduce the time required to solve the entire problem. Tasks can be of arbitrary size, but once defined, they are the smallest units of concurrency. The process of expressing a computation as a collection of tasks is called ***decomposition***. 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***.

Consider the problem of multiplying two dense matrices A and B to yield a matrix C. Figure 4.1 shows a decomposition of this problem into four tasks. Each matrix is considered to be composed of four blocks or submatrices defined by splitting into half each dimension of the matrix. The submatrices of C are then independently computed by four tasks as the sums of the appropriate products of submatrices of A and B.

In the above example, the four tasks are closely related to the decomposition of the matrices (*i.e.*, the data) themselves. Data decomposition is the topic of Section 4.2.2 to appear later in this chapter. 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 4.2 shows two other decompositions of matrix multiplication, each into eight tasks, corresponding to the same data decomposition as used in Figure 4.1.

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: $C_{2,2} = C_{2,2} + A_{2,2}B_{2,2}$

Figure 4.2 Two examples of decomposition of matrix multiplication into eight tasks.

Note that all four tasks in Figure 4.1 are independent and can be performed all together or in any sequence. However, that is not the case with the decompositions shown in Figure 4.2. Each of the even numbered tasks in both decompositions of Figure 4.2 can be performed only after the immediately preceding odd numbered task is complete. The reason is that each even numbered task shares intermediate data (*i.e.*, partial results in C submatrices) with the preceding odd numbered task. An abstraction used to express such dependencies among tasks and their relative order of execution is known as a ***task graph***. A task 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. The task graphs for the matrix multiplication problem described above are fairly straightforward (Problem 4.1).

ID#	Model	Year	Color	Dealer	Price
4523	Accord	2000	Blue	MN	$19,000
3476	Taurus	2000	Green	IL	$18,000
7623	Camry	1999	Black	NY	$21,000
9834	Civic	1997	Black	MN	$10,000
6734	Accord	1999	Green	AR	$18,000
5342	Altima	1999	Black	FL	$18,000
3845	Maxima	1999	Blue	TX	$22,000
8354	Malibu	2000	Black	ID	$16,000
4395	Accord	1999	Red	CA	$20,000
7352	Accord	2000	Red	SD	$17,000

Table 4.1 A database storing information about used vehicles

To study a more interesting task graph, let us consider the computations performed on the vehicle relational database, shown in Table 4.1, in processing the following query:

(MODEL="Accord" AND YEAR="1999") AND (COLOR="Green" OR COLOR="Black")

This query looks for all 1999 Accords whose color is either Green or Black. On a relational database, this query will be processed by creating a number of intermediate tables. One possible way is to first create the following four tables: A table containing all Accords, a table containing all 1999-model cars, a table containing all green-colored cars, and a table containing all black-colored cars. Next, the computation will proceed by combining these tables via what is known as a database-join operation. In particular, it will compute the intersection of the Accord-table with the 1999-model year table, to construct a table of all 1999-model Accords. Similarly, it will join the green- and black-colored tables to compute a table storing all cars whose color is either green or black. Finally, it will compute the intersection of the table containing all the 1999 Accords with the table containing all the green or black vehicles, and return the desired list. These computations can be visualized by a task graph as shown in Figure 4.3. Each node in this figure is a task that corresponds to an intermediate table that needs to be computed. Arrows between nodes indicate dependencies between the tasks. For example, before we can compute the table that corresponds to the 1999 Accords, we must first compute the table of all the Accords and a table of all the 1999-model cars.

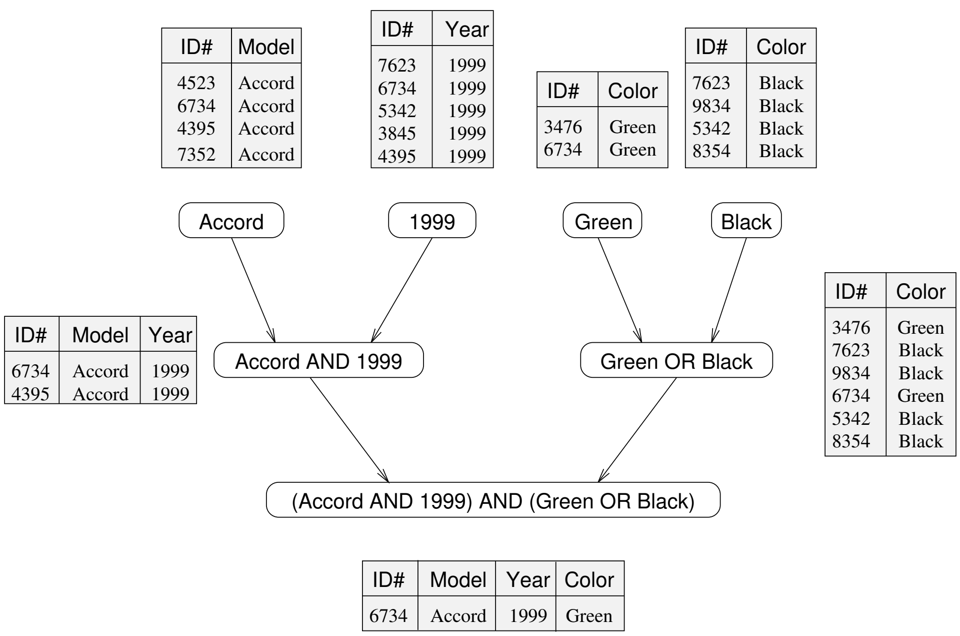


Figure 4.3 The different tables generated by processing the query and their dependencies.

We now introduce the problem of LU factorization, which too yields non-trivial task graphs. In its simplest form, the LU factorization algorithm factors a nonsingular square

```
1.  procedure COL_LU (A, b, y)
2.  begin
3.      for k := 0 to n − 1 do
4.      begin
5.          for j := k to n − 1 do
6.              A[j, k] := A[j, k]/A[k, k];
7.          endfor;
8.          for j := k + 1 to n − 1 do
9.              for i := k + 1 to n − 1 do
10.                 A[i, j] := A[i, j] − A[i, k] × A[k, j];
11.             endfor;
12.         endfor;
13.     endfor;
14. end COL_LU
```

Algorithm 4.1 A serial column-based algorithm to factor a nonsingular matrix A into a lower-triangular matrix L and an upper-triangular matrix U. Matrices L and U share space with A. On Line 10, $A[i, j]$ on the left side of the assignment is equivalent to $L[i, j]$ if $i > j$; otherwise, it is equivalent to $U[i, j]$.

matrix A into the product of a lower triangular matrix L with a unit diagonal and an upper triangular matrix U. Program 4.1 shows the serial algorithm. If A is an $n \times n$ matrix with rows and columns numbered from 0 to $n - 1$, then the factorization process consists of n major steps. In step k, first, the partial column $A[k + 1 : n - 1, k]$ is divided by $A[k, k]$. Then, the outer product $A[k + 1 : n - 1, k] \times A[k, k + 1 : n - 1]$ is subtracted from the $(n-k-1) \times (n-k-1)$ submatrix $A[k+1 : n-1, k+1 : n-1]$. The algorithm starts with $k = 0$ and terminates after $k = n - 1$. In practice, separate arrays are not used for L and U and A is modified to store L and U in its lower and upper triangular parts, respectively (the 1's on the principal diagonal of L are implicit and the diagonal entries actually belong to U after factorization).

Like most matrix algorithms, including the matrix-multiplication example considered earlier, LU factorization can be formulated in terms of block matrix operations In this 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 4.3. Block versions of matrix algorithms are often used to aid decomposition. In Figure 4.4, we show the decomposition of LU factorization into 14 tasks based on a 2-dimensional partitioning of the matrix A into nine blocks $A_{i,j}$, $1 \leq i, j \leq 3$. The blocks of A are modified into corresponding blocks of L and U as a result of factorization. The diagonal blocks of L are lower triangular submatrices with unit diagonals and the diagonal blocks of U are upper triangular submatrices.

Figure 4.5 shows the task graph for LU factorization corresponding to the decompo-

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

1: $A_{1,1} \rightarrow L_{1,1}U_{1,1}$	6: $A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$	11: $L_{3,2} = A_{3,2}U_{2,2}^{-1}$
2: $L_{2,1} = A_{2,1}U_{1,1}^{-1}$	7: $A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$	12: $U_{2,3} = L_{2,2}^{-1}A_{2,3}$
3: $L_{3,1} = A_{3,1}U_{1,1}^{-1}$	8: $A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$	13: $A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$
4: $U_{1,2} = L_{1,1}^{-1}A_{1,2}$	9: $A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$	14: $A_{3,3} \rightarrow L_{3,3}U_{3,3}$
5: $U_{1,3} = L_{1,1}^{-1}A_{1,3}$	10: $A_{2,2} \rightarrow L_{2,2}U_{2,2}$	

Figure 4.4 A decomposition of LU factorization.

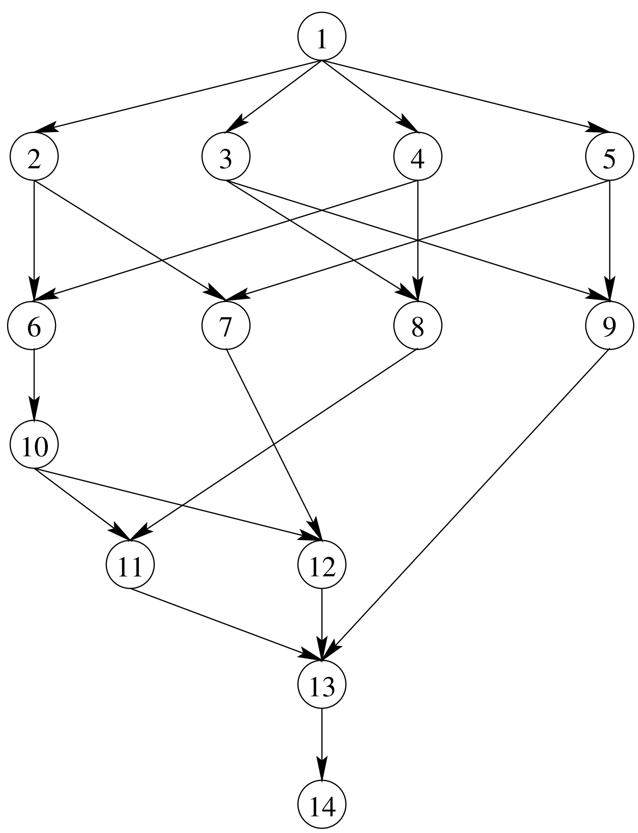


Figure 4.5 A task graph for LU factorization with the decomposition shown in Figure 4.4.

	Mapping I	Mapping II
Process	Tasks	Tasks
P_0	1, 2, 5, 7, 12, 13, 14	1, 2, 6, 10, 11, 13, 14
P_1	4, 6, 10, 11	5, 7, 12
P_2	3, 8, 9	4, 8
P_3		3, 9
Time step	Tasks	Tasks
1	1	1
2	2, 3, 4	2, 3, 4, 5
3	5, 6, 8	6, 7, 8, 9
4	7, 9, 10	10
5	11, 12	11, 12
6	13	13
7	14	14

Figure 4.6 Two mappings of LU factorization with decomposition shown in Figure 4.4. Mapping I utilizes three processes and Mapping II uses four processes.

sition of Figure 4.4. The parallel program starts with Task 1. The completion of Task 1 makes the four tasks numbered 2 through 5 executable, and so on. The parallel algorithm terminates when Task 14 is finished.

4.1.2 Processes and Mapping

The tasks into which a problem is decomposed run on physical processors. However, for reasons that we shall soon discuss, we would 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, besides computing and reading and writing data, 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***.

As an example of a simple mapping, the four tasks in Figure 4.1 could be mapped onto four different processes running simultaneously. A possible mapping for the decompositions of Figure 4.2 onto four processes may assign Tasks 1 and 2 to process P_0, Tasks 3 and 4 to process P_1, Tasks 5 and 6 to process P_2, and Tasks 7 and 8 to process P_3. Note that, in this case, a maximum of four processes can be employed usefully, although the total number of tasks is eight. This is because the dependencies among the tasks, as prescribed by the task graph, permit a maximum of only four tasks to be executable at any given time.

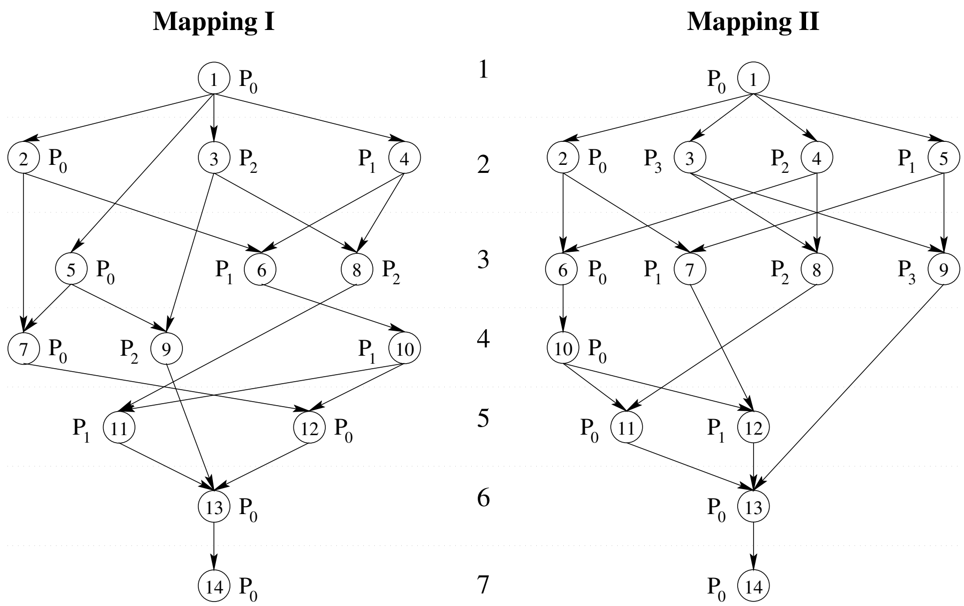


Figure 4.7 Two mappings corresponding to the task graph shown in Figure 4.5.

The maximum number of tasks that can be executed simultaneously in a parallel program at any given time is known as its ***degree of concurrency***. The degree of concurrency is equal to the maximum number of processes that can be employed for solving a problem such that no process is idle during at least some fraction of the duration of the execution of the parallel program. The maximum degree of concurrency for LU factorization with the decomposition shown in Figure 4.4 is four.

A concept related to degree of concurrency is that of a ***critical path***. In a task 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, and the number of nodes along this path is known as the ***critical-path length***. The ratio of the total number of tasks in a decomposition to the critical-path length is an lower bound on the degree of concurrency. Therefore, a shorter critical path favors a higher degree of concurrency. When the tasks along the critical path are of varying sizes, critical-path length is often defined in terms of time as the sum of the execution times of the tasks along the critical path. This time is a lower bound on the total execution time of a parallel program. Although the degree of concurrency and the critical path are functions of the decomposition and the structure of the task graph, they play an important role in the selection of a good mapping for a parallel program.

Figures 4.6 and 4.7 show two mappings of the decomposition of Figure 4.4, one on three processes and the other on four. For the sake of simplicity, let us assume that each task takes a unit amount of time[1]. If a task is executed as soon as the task graph of Figure 4.5 permits, then both the mappings of Figure 4.6 require seven time steps to complete the factorization. This shows that, in general, increasing the number of processes does not always result in a shorter completion time for a parallel program, even if other overheads of parallelization are not take into account. Since the critical path-length in this case is seven, the mapping that uses fewer processes to complete the program in seven steps is more efficient.

The mapping of tasks onto processes plays an important role in determining how efficient the resulting parallel program is. Even though the degree of concurrency is determined by the decomposition, it is mapping that determines how much of that concurrency is actually utilized. However, as we will discuss in greater detail later in this and other chapters, there are factors other than concurrency that affect parallel program performance, and hence, the choice of a good mapping. The most important among these factors is the time that tasks may spend interacting with each other. In the examples considered so far, we have only emphasized on the portion of the computation that a task is responsible for. Typically, in addition to performing computations, tasks exchange information with other tasks in a parallel program. Such information that is exchanged between tasks includes intermediate results, shared input data, synchronization information, etc. Often, and particularly in a distributed-memory paradigm, most of the input and output data for a parallel program is partioned and distributed among multiple memory banks in order to effectively utilize the aggregate memory in the parallel computer. However, tasks may require access to portions of data that reside in different memory banks. A systematic exchange of data among tasks must ensure that each task has all the data that it requires to perform the computations assigned to it.

Consider the problem of computing the matrix-vector product $y = Ab$ where A is a sparse matrix and b is a dense vector. 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. One possible way of decomposing this computation is to partition the output vector y and have each task compute an entry in it. Figure 4.8(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 location. In the message-passing paradigm, with the ownership of $b[i]$, Task i also inherits the responsibility to send $b[i]$ 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 $b[4]$, $b[5]$, and $b[8]$ to perform its own computation. Such interactions consume time and are a cause of overhead in a parallel program. In the message-passing paradigm, these interactions take the form of time consuming sending and receiving of messages. In the shared-address-space paradigm, the overhead may result from contention for shared

[1]In practice, for a block size $b > 1$, Tasks 1, 10, and 14 require about $2/3b^3$ arithmetic operations; Tasks 2, 3, 4, 5, 11, and 12 require about b^3 operations; and Tasks 6, 7, 8, 9, and 13 require about $2b^3$ operations.

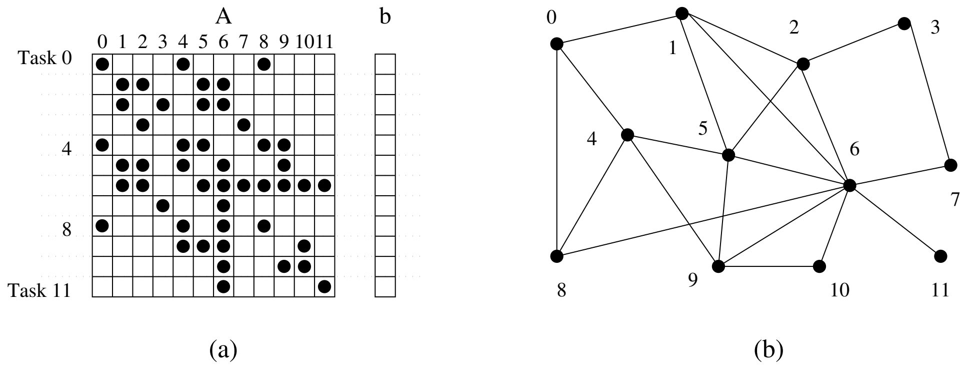


Figure 4.8 A decomposition for sparse matrix-vector multiplication and the corresponding task interaction graph. In the decomposition Task i computes $\sum_{0 \le j \le 11, A[i,j] \ne 0} A[i, j].b[j]$.

hardware resources during simultaneous accesses to the same data by multiple tasks.

The inter-task interaction overhead may be reduced if heavily interacting tasks are mapped onto the same process so that the data does not require an actual physical transfer. An abstraction that is often used to capture the interaction among tasks and aid in devising a mapping that minimizes inter-process interaction is known as a ***task interaction graph***. In a task interaction graph, the nodes represent tasks. A pair of nodes is connected by an edge if that pair of nodes interacts at any time during the execution of the parallel program. Figure 4.8(b) shows a task interaction graph for the instance of sparse matrix multiplication problem shown in Figure 4.8(a). 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.

4.1.3 Processes Versus Processors

In the context of parallel program 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 a majority of 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 use as many processes as the number of physical CPU's on the parallel computer. However, often, a higher level of abstraction is required to express a parallel algorithm; especially, if it is a complex algorithm with multiple stages or with different types of parallelisms. For example, consider the decomposition of LU factorization into 14 tasks in Figure 4.4. In this decomposition, there are three types of tasks. The first kind is factorization; *i.e.*, Tasks 1, 10, and 14. The second kind is multiplying a submatrix with the inverse of a triangular matrix; *i.e.*, Tasks 2, 3, 4, 5, 11, and 12. The third kind is adding the product of two matrices to a third matrix; *i.e.*, Tasks 6, 7, 8, 9, and 13. Each of these tasks can themselves be performed in parallel. One possible parallel design of an LU factorization algorithm could consist of first performing a coarse-grain decomposition and mapping the tasks into relatively few processes, and then treating each task as a parallel program in itself with its own decomposition and mapping. In this case, the number of processes at any of the two levels of parallelism is not likely to be the same as the number of physical processors available.

Treating processes and processors separately is also useful when designing parallel program 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 CPU's. Consider implementing matrix multiplication on a parallel computer like this. The best way to design a parallel program 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 CPU's of a node.

4.2 Decomposition Techniques

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 graph. A good decomposition and mapping combination should be such that it minimizes the process idle time and inter-task interaction while achieving the completion-time objectives of parallelization. Note that 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. Therefore, a decomposition must lead to a sufficiently high degree of concurrency so that the resulting tasks can be mapped onto the available processes. Often the goals of higher concurrency and lower interaction conflict with each other. 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, they 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 problems. On the other hand, speculative- and exploratory-decomposition techniques more of a *special purpose* nature because they apply to specific classes of problems.

4.2.1 Recursive Decomposition

Recursive decomposition is a method for inducing concurrency in problems that are often solved using the divide-and-conquer paradigm. In this paradigm, 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 paradigm results in natural concurrency, as different subproblems can be solved concurrently.

Consider the problem of sorting a sequence A of n elements using the popular 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. The subsequences A_0 and A_1 are then sorted obtaining A'_0 and A'_1, respectively, and the sorted sequence A' is obtained by appending A'_1 to sequence A'_0. Each one of the subsequences A_0 and A_1 are sorted by recursively calling quicksort. Each one of these recursive calls further partitions the sequences. This is illustrated in Figure 4.9 for a sequence of 12 numbers. The recursion terminates when each subsequence contains only a single element.

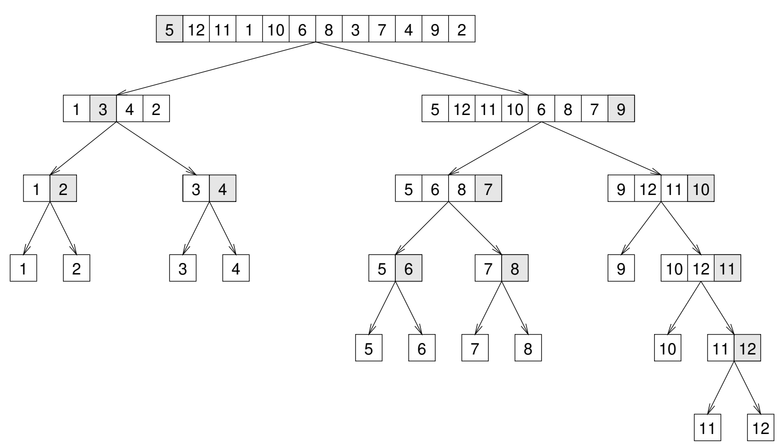


Figure 4.9 The quicksort task graph based on recursive decomposition for sorting a sequence of 12 numbers.

If partitioning a given subsequence is defined to be a task, then Figure 4.9 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 increases as we move down the tree.

Recursive decomposition can also be applied to extract concurrency in problems for which the algorithm used to solve this problem serially is not necessarily recursive. 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 the following code.

```
Minimum(A, n)
  min = A[0]
  for (i=1; i<n; i++)
    if (A[i] < min)
      min = A[i]
  return min
```

Looking at this program, we can easily see that this serial algorithm exhibits no concurrency.

One way of restructuring the computations is to formulate the problem as a divide and conquer algorithm, and then use recursive decomposition to extract concurrency. The divide and conquer formulation of our minimum finding algorithm is shown in the following code fragment:

```
Minimum(A, n)
  if (n == 2)
    lmin = A[0]
    rmin = A[1]
  else
    lmin = Minimum(A, n/2)
    rmin = Minimum(A+n/2,n-n/2)
  end if
  if (lmin < rmin)
    return lmin
  else
    return rmin
```

That is, 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 graph for this problem. Figure 4.10 illustrates such a task graph for finding the minimum of eight numbers where each task is assigned the work of finding the minimum of two numbers.

4.2.2 Data Decomposition

Data decomposition is a powerful and one of the most 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 (or the domain) 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

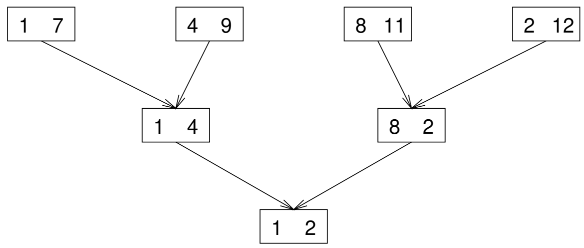


Figure 4.10 The task graph for finding the minimum number in a sequence of eight numbers. Each node in the tree represents the task of finding the minimum of the two numbers in the node.

these tasks perform on different data partitions are usually similar (*e.g.*, matrix multiplication) or are chosen from a small set of operations)*e.g.*, LU factorization). 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 good (or easiest) computational decomposition.

Partitioning Output Data When the output of an algorithm is a large data structure with a predictable size and memory-location, and when portions of this data structure can be computed independently of others, then a decomposition of the problem may be induced by a partioning of this output data structure.

For example, consider the problem of matrix-multiplication discussed in Section 4.1.1. The decomposition shown in Figure 4.1 is based on partitioning the output matrix C into four submatrices and each of the four tasks computes one of these submatrices.

As another example, consider the problem of rendering a scene using ray-tracing. Ray-tracing is a popular algorithm used in computer graphics to achieve a realistic rendering of a scene. In ray-tracing, the color value for each pixel in the image is determined by "shooting" a ray starting from the viewpoint and passing through the center of that pixel. In its simplest form, as soon as the ray hits an object the contributions of the different light sources is computed at the point of intersection, and it is used as the color of the pixel. However, in order to handle shadows, reflection, and refraction, additional rays are spawned from the point of intersection. In particular, to calculate shadows, additional rays are fired from the point of intersection to each one of the light sources, as illustrated in Figure 4.11. If any of these shadow rays intersects any objects along the way, then the object is in shadow at that point and the algorithm ignores the contribution of the shadow ray's light source. Similarly, to include specular reflection and refractive transparency, the algorithm spawns reflection rays and refraction rays from the point of intersection, depending on the material properties of the intersected object. Each of these additional rays may in turn spawn additional shadow, reflective, and refractive rays. For example, the second ray shown in Figure 4.11 first hits the pyramid, then its reflective ray hits the cube whose reflective ray hits the cylinder.

The output of the ray-tracing algorithm is a two-dimensional array corresponding to the

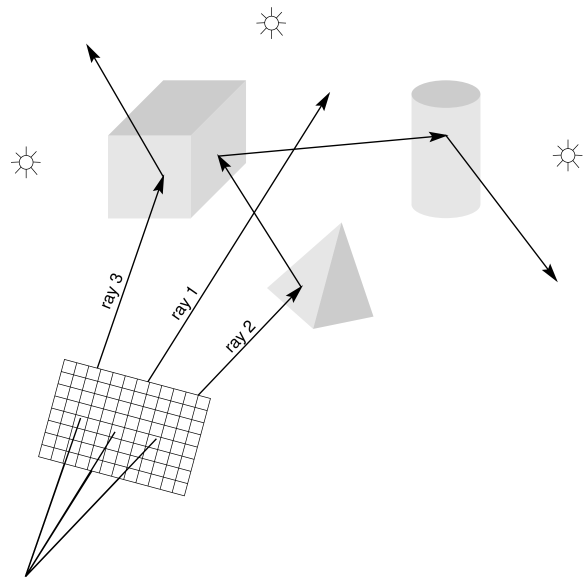
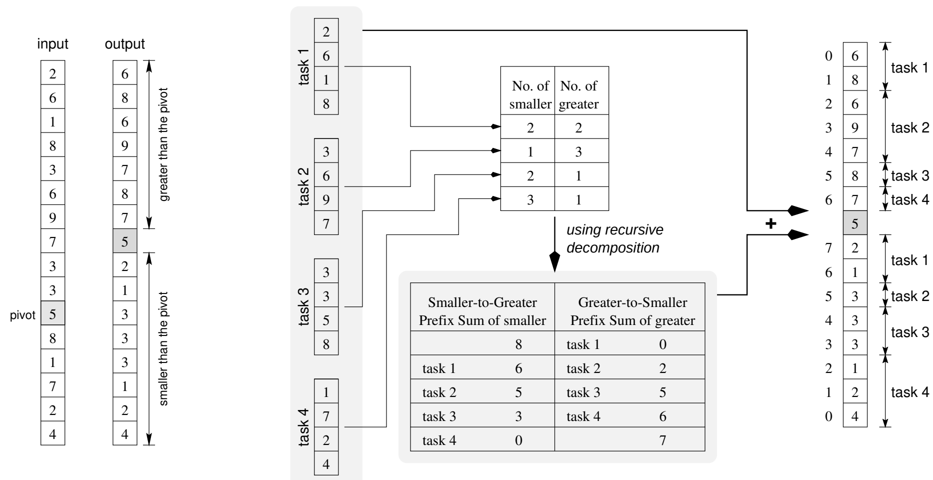


Figure 4.11 An example of ray-tracing showing that the amount of computation performed for each ray is different.

size of the image. Each entry of this array stores the value of the corresponding pixel, computed by a ray passing through that pixel. Since the computation of color values of each pixel is independent of the computation for other pixels, we can decompose the computations performed by the algorithm by partitioning the array of pixels.

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, for 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 as follows. A task is created for each partition of the input data. This task performs as much computation as possible using this 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.

A problem in which decomposition of input is a natural method for deriving concurrency is that of partitioning an array of numbers into two parts such that one part contains numbers smaller than a specified element within the array (called *pivot*) and the other part contains the remainder. To simplify the discussion, we assume that the pivot element is unique; *i.e.*, all other elements of the array are different than from the pivot. This problem is solved in two passes over the data. In the first pass, the number of elements to be put in each partition is counted and the location of the pivot in the output array is determined. In the second pass, the elements are moved from the input array to their respective partitions in the output. Figure 4.12(a) shows an example problem for an array of 16 numbers.



(a) Partitioning an array around a pivot element

(b) Using data decomposition of the input to perform this computation in parallel

Figure 4.12 Partitioning a list around a pivot element.

While solving this problem in parallel, it is not possible to decompose the computations by partitioning the output sequence, as individual elements cannot be independently computed as functions of the input efficiently. However, it is possible to decompose this computation by partitioning the input array and assigning each input partition to a different task. This decomposition is illustrated in Figure 4.12(b), in which the input sequence is partitioned among four tasks, each responsible for four elements of the input sequence. Now each task compares elements assigned to it against the pivot to determine how many of them are smaller and how many of them are greater than the pivot element. Once this is done, then it is possible for each task to partition their local lists into two parts—one smaller and the other greater than the pivot. But what we need is a global partition of the entire input array. To achieve this, each task needs to determine the locations in the output array where it can copy the two sets of elements. This can be done using the prefix sum operation. The prefix sum of an array A of numbers is another array S such that $S[i] = \sum_{j=0}^{i} A[j]$. A parallel algorithm for the computing prefix sums is discussed in detail in Section **??** and involves a recursive decomposition. The entire parallel algorithm for the array-partitioning problem is illustrated in Figure 4.12. We compute two prefix sums, one for the elements that are smaller than the pivot and one for the elements that are greater than the pivot. The first prefix sum computes the sum starting from the smallest-numbered task whereas the second prefix sum starts from the highest-numbered task. As shown in the figure, these prefix sums yield the starting indices for the output of each task belonging to each partition. Once each task knows the locations in the target array that it needs to copy its smaller and greater elements into, all tasks can proceed concurrently.

In some algorithms in which it is possible to partition the output data, partitioning of input data can offer additional ways to obtain concurrency. For example, 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 M. Our goal is to find the number of times that each itemset in I appears in all the transactions. Figure 4.13(a) shows an example of this type of computation. Our database consists of ten transactions, and we are interested in computing the frequency of the eight itemsets shown in the second column. The actual frequency of these itemsets in the database are shown in the third column. For instance, itemset {D, K} appears two times, once in the second and once in the ninth transaction.

In this problem, we see that we have two inputs, the sets T and I and one output, an array of size m that stores the frequency of each itemset in I. Any of these data could be partitioned to induce a decomposition of the overall computations. One possibility is to partition the set of transactions T into equal size parts, as shown in Figure 4.13(b). Task associated with each partition now will be responsible for computing the frequencies of all the itemsets in I based solely upon the transactions it owns. The final result will be obtained by adding these frequencies over all the partitions. Another possibility is to partition the itemsets I, into an equal number of parts, as shown in Figure 4.13(c). Task associated with each partition now will compute the frequencies of only the itemsets that it owns. Note also, that this is exactly the same computational decomposition that we would have obtained if we were to partition the output array.

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 input data can sometimes lead to higher concurrency than partitioning input or output data. Often, the intermediate data is 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. Figure 4.14 shows a decomposition for the matrix multiplication problem $C = A.B$. The decompositions induced by a 2×2 partitioning the output matrix C, as shown in Figures 4.1 and 4.2, have a degree of concurrency of four. The decomposition shown in Figure 4.14 has a degree of concurrency of eight because the first eight tasks can all be performed in parallel. This decomposition is induced by a partitioning of the intermediate 3-dimensional matrix D into eight parts. Note that all elements of D are computed implicitly in the original algorithm, but are not explicitly stored. By restructuring the original algorithm and by explicitly storing D, we have been able to devise a decomposition

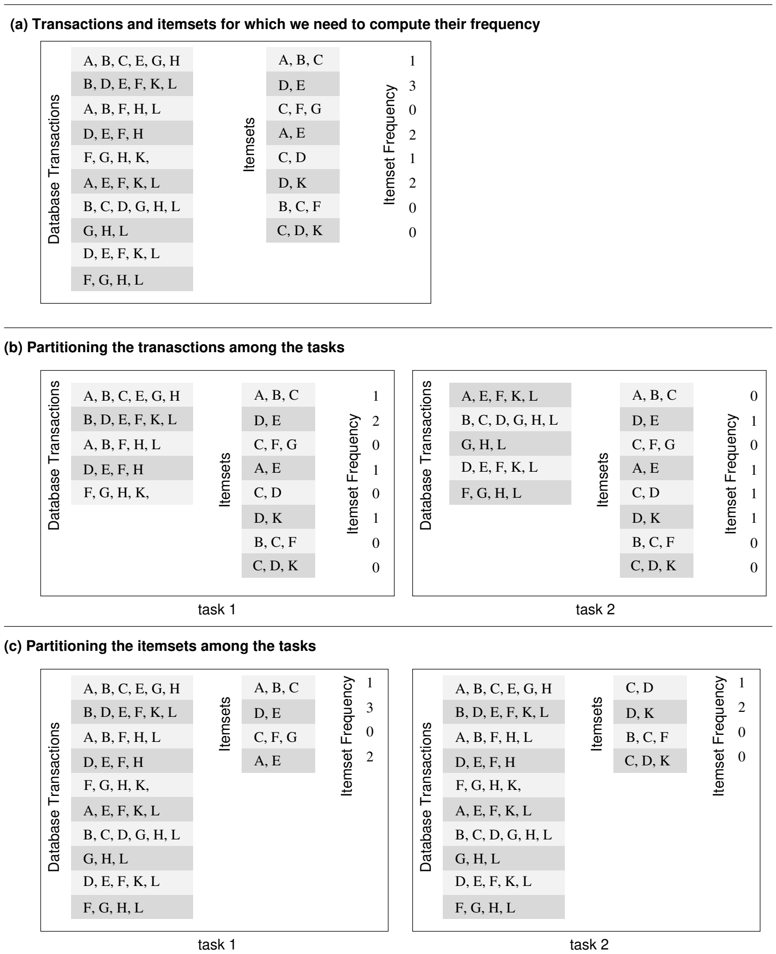


Figure 4.13 A sample transaction database and itemsets.

with higher concurrency. This, however, has been achieved at the cost of extra aggregate memory usage.

The Owner-Computes Rule The above decomposition schemes are examples of a widely used method of inducing a computational decomposition from a data partitioning called the ***owner-computes*** rule. The idea behind this rule is that each partition performs all the computations involving data that it owns. Of course, 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 partition the input data, then the owner-computes rule means that we perform 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 we perform all the

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

Task 01: $D_{1,1,1} = A_{1,1}B_{1,1}$
Task 02: $D_{2,1,1} = A_{1,2}B_{2,1}$
Task 03: $D_{1,1,2} = A_{1,1}B_{1,2}$
Task 04: $D_{2,1,2} = A_{1,2}B_{2,2}$
Task 05: $D_{1,2,1} = A_{2,1}B_{1,1}$
Task 06: $D_{2,2,1} = A_{2,2}B_{2,1}$
Task 07: $D_{1,2,2} = A_{2,1}B_{1,2}$
Task 08: $D_{2,2,2} = A_{2,2}B_{2,2}$
Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$
Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$
Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$

Figure 4.14 A decomposition of matrix multiplication based on partitioning the intermediate three-dimensional matrix.

computations for these data.

4.2.3 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.

As an example of exploratory decomposition, consider the 15-puzzle problem. The 15-puzzle consists of 15 tiles numbered 1 through 15 and one blank tile placed in a 4×4 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 4.15

illustrates sample initial and final configurations and a sequence of moves leading from the initial configuration to the final configuration.

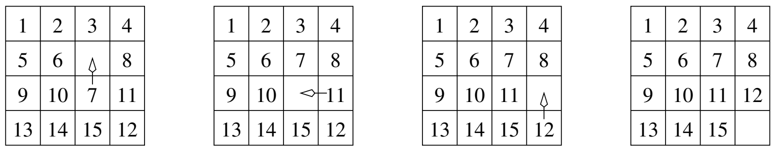


Figure 4.15 A 15-puzzle problem instance: (a) initial configuration; (b) final configuration; and (c) a sequence of moves leading from the initial to the final configuration.

The 15-puzzle is typically solved using tree-search techniques. Starting from the initial configuration, all possible successors are generated. A configuration may have 2, 3, or 4 possible successor configurations. The task of finding a path from initial to final configuration now translates to finding a path from one of these newly generated nodes to the final configuration. Since one of these newly generated nodes must be closer to the solution by one move (if a solution exists), we have made some progress towards finding the solution.

To solve this problem in parallel, we first serially generate a few levels of configurations starting from the initial configuration until the search tree has a sufficient number of leaf nodes (*i.e.*, configurations of the 15-puzzle). For example, starting from the initial state, up to four successor states can be generated—one corresponding to the occupation of the empty slot by each of its four possible neighbors. Similarly, each of these states can be further expanded to generate additional states. This process continues until a preset number of states have been generated. Now each state 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 4.16 illustrates one such decomposition into four tasks in which task 3 finds the solution.

Note that even though exploratory decomposition may appear similar to data decomposition (the search space can be thought as being the data that gets 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 are terminated as soon as an overall solution is found. Hence, the portion of the search space searched (and hence, the aggregate amount of work performed) by a parallel formulation can be much different than that searched by a serial algorithm. The work performed by the parallel formulation can either be smaller or greater than that performed by the serial algorithm. For example, consider a search space that has been partitioned into four concurrent tasks. If the solution lies right at the beginning of the search space corresponding to task 3, 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

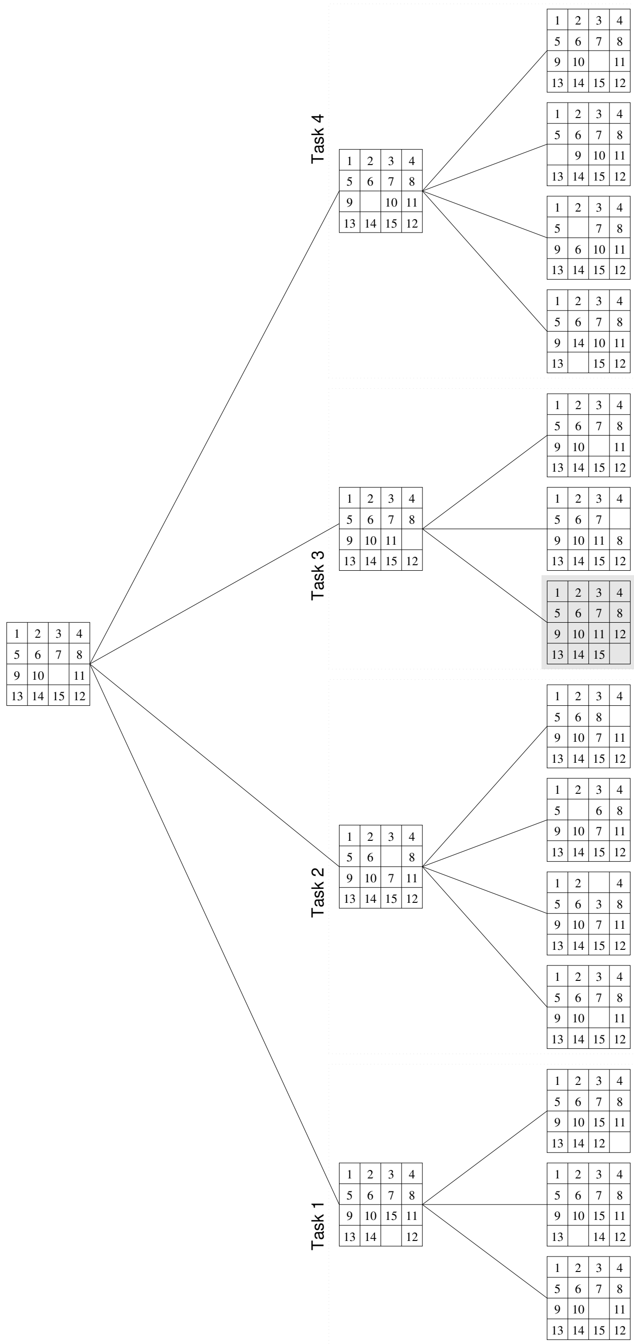


Figure 4.16 The states generated by the 15-puzzle problem.

towards the end of the search space corresponding to task 1, then the parallel formulation will perform almost four times the work of the serial algorithm and will yield no speedup.

4.2.4 Speculative Decomposition

Speculative decomposition is used when a program may take one of many possible computationally significant branches depending on the output of another computation. In this situation, while one task is performing the computation whose output is used in deciding the next computation, several tasks can concurrently embark upon the multiple computations of the next stage. This scenario is similar to evaluating all the branches of a *switch* statement in C in parallel before the input for the *switch* is available. When the input for the *switch* has finally been computed, the computation corresponding to the correct branch is used while that corresponding to the other branches is discarded. The parallel run time is smaller than the serial tun 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. 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.

Speculative decomposition is different from exploratory decomposition in the following way. In speculative decomposition, the serial algorithm would strictly perform only one of tasks at a speculative stage because when it reaches the beginning of that stage, it knows exactly which branch to take. On the other hand, in exploratory decomposition, the serial algorithm too may explore different alternatives one after the other, because it does not know which branch may lead to the solution. In speculative decomposition, the parallel program always performs more aggregate work than the serial algorithm. In exploratory decomposition, the parallel program may perform more or less aggregate work than the serial algorithm.

4.2.5 Hybrid Decomposition

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. In fact, for many problems, it is necessary to combine different decomposition techniques to obtain a good parallel algorithm.

For example, in Section 4.2.1, we used 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 4.2.2. The resulting hybrid decomposition that combines recursive decomposition and the input data decomposition leads to a highly concurrent formulation of quicksort.

If the computation is structured into multiple stages, then it is sometimes necessary to apply different types of decomposition in different stages. For example, Figure 4.14 shows a two-stage formulation of matrix-matrix multiplication. The first stage of this formulation involves multiplication of submatrices of the original matrices. The second stage involves addition of the intermediate submatrices. The example shows a $2 \times 2 \times 2$ partition of D and the addition of only two pairs of submatrices $D_{1,*,*}$ and $D_{2,*,*}$. In general, the intermediate data could have a $k \times i \times j$ partition, in which case, the second stage would involve an addition of k sets of i, j submatrices. Each of these additions can be performed using a recursive decomposition, as shown in Figure 4.10 for $k = 4$.

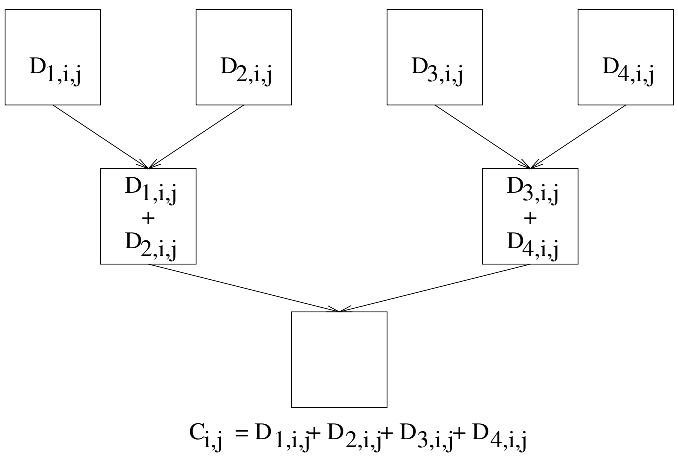


Figure 4.17 A recursive decomposition for submatrix addition.

4.3 Characteristics of Tasks and Interactions

The various decomposition techniques described in the previous section allow us to identify the concurrency that is available in a problem and decompose it into tasks that can be executed in parallel. The next step in the process of designing a parallel algorithm is to take these tasks and assign (*i.e.*, map) them onto the available processes. While devising a mapping scheme to construct a good parallel algorithm, we often take a cue from the decomposition. The nature of the tasks and the interactions between them has a bearing on the mapping. In this section, we shall discuss the various properties of tasks that affect the choice of a good mapping.

4.3.1 Characteristics of Tasks

The following three characteristics of the tasks have a large influence on the suitability of a mapping scheme.

Task Generation The tasks that constitute a parallel algorithm may either be generated statically or dynamically. ***Static task generation*** refers to the scenario where all the tasks are known before the algorithm starts executions. Naturally, the task graph too is static in this case; *i.e.*, the entire task graph is known a-priori. Data decomposition usually leads to static task generation. Examples of data decomposition leading to a static task generation include matrix-multiplication and LU factorization discussed earlier. Certain recursive and speculative decompositions can also lead to a static task graph. Finding the minimum of a list of numbers (Figure 4.10 is an example of static recursive task graph.

Certain decompositions lead to a ***dynamic task generation*** during the execution of the algorithm. Although the high level rules or guidelines governing tasks generation are known as a part of the algorithm, the actual tasks and the task graph is not explicitly available a-priori. Recursive decomposition can lead to dynamic task generation. For example, consider the recursive decomposition in quicksort (Figure 4.9). The tasks are generated dynamically, and the size and shape of the task tree is determined by the values in the input array to be sorted. An array of the same size can lead to task graphs of different shapes and sizes (in terms of the number of tasks).

Exploratory decomposition can be formulated to generate tasks either statically or dynamically. For example, consider the 15-puzzle problem[2] discussed in Section 4.2.3. We can define a decomposition static task graph as follows. First, a task starts with the initial state and expands the search tree in a breadth-first manner until a predefined number of states are obtained. Now a set of subsequent independent tasks explore each of the state resulting from the first task until at least one of them finds a solution. A different decomposition that generates tasks dynamically would be one in which a task takes a state as input, expands it through a predefined number of steps of breadth-first search and spawn a new task to perform the same computation on each of the resulting states (unless it has found the solution, in which case the program terminates).

Task Sizes The complexity of mapping schemes often depends on whether or not the tasks are uniform. A set of tasks are said to be ***uniform*** if each one requires roughly the same amount of time. On the other hand, if the amount of time required by the tasks varies significantly, then they are said to be ***non-uniform***. For example, the tasks in the decompositions for matrix multiplication shown in Figures 4.1 and 4.2 would be considered uniform. On the other hand, the tasks in the decomposition for LU factorization in Figure 4.4 are non-uniform.

[2] In this chapter, we have used the 15-puzzle example as an aid to exposition of certain concepts. We do not describe the best algorithm to solve this problem, which is beyond the scope of this chapter.

Knowledge of Task Sizes The second characteristic that influences the choice of the mapping scheme is the knowledge of the task size; *i.e.*, the amount of time that is required to complete them. If the size of all the tasks is known, then this information can often be used in mapping of tasks to processes. For example, in the various decompositions for matrix multiplication discussed so far, the computation time for each task is known before the parallel program starts. On the other hand, in the ray tracing example in Section 4.2.2, although the task generation is static, the size of each task is unknown because different rays may require different amounts of time to trace depending on their paths.

Size of Data Associated with Tasks The third important characteristic of a task is the size of data associated with it. The reason this is an important consideration for mapping is that the data associated with a task must be available to the process performing that task and the size and the location of this data may dictate which process may perform which task without incurring excessive data-movement overheads.

Different types of data associated with a task may have different sizes. For instance, the input data data may be small but the output may be large, or vice-versa. For example, the input to a task in the 15-puzzle problem may be just one state of the puzzle. This is a small input to the potential amount of computation that may be required to find a sequence of moves from this state to a solution state. In the problem of computing the minimum of a sequence, the size of the input is proportional to the amount of computation, but the output is tiny. In the parallel formulation of the quick sort, the size of both the iput and the output data is of the same order as the time needed to solve the task.

4.3.2 Characteristics of Inter-task Interactions

In any non-trivial parallel program, tasks need to interact with each other to share data, work, or synchronization information. Different parallel algorithms require different types of interactions among concurrent tasks. The nature of these interactions make them more suitable for certain programming paradigms and/or mapping schemes, and less suitable for others. The types of inter-task interactions can been described along different dimensions, each dimension corresponding to a distinct characteristic of the underlying computations.

Static versus Dynamic One way of classifying the type of interactions that take place among concurrent tasks is to consider whether or not these interactions have a ***static*** or ***dynamic*** pattern. An interaction pattern is static if for each task, the interactions happen at predetermined times, and if the set of tasks to interact with at these times is known prior to the execution of the algorithm. An interaction pattern is dynamic if the timing of interactions or the set of tasks to interact with cannot be determined prior to the execution of the algorithm; *i.e.*, the task interaction graph is not known a-priori.

Static interactions can be programmed easily in message-passing paradigm, but dynamic interactions are harder to program. The reason is that interactions in message-passing require active involvement of both interacting tasks—the sender and the receiver

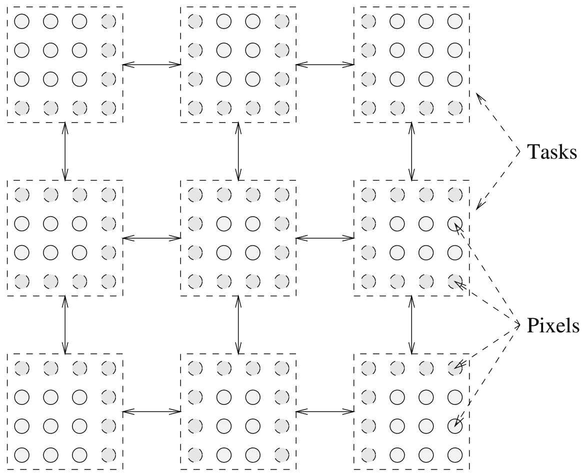


Figure 4.18 The regular two-dimensional task interaction graph for image dithering. The pixels with dotted outline require color values from the boundary pixels of the neighboring tasks.

of information. The unpredictable nature of dynamic iterations makes it hard for both partners to participate in the interaction at the same time. Shared-address space programming can code both types of interactions equally easily.

Regular versus Irregular Another way of classifying the interactions is based upon their spatial structure. An interaction pattern is considered to be ***regular*** if it has some structure that can be exploited for efficient implementation. On the other hand, an interaction pattern is called ***irregular*** if no such regular pattern exists.

An example of a parallel formulation with regular interaction pattern is the problem of image dithering. In this computation, the color of each pixel is determined as the weighted average of its original value and the values of its neighboring pixels. We can easily decompose this computation, by breaking the image into square regions and use a different task to dither each one of these regions. Note that each task needs to access the pixel values of the region assigned to it as well as the values of the image surrounding its region. Thus, if regard the tasks as nodes of a graph with an edge linking a pair of interacting tasks, the resulting pattern in a two-dimensional mesh, as shown in Figure 4.18.

For an example of irregular interaction, consider the problem of computing the product y of a sparse matrix A with a dense vector b. One possible way of decomposing this computation is to partition the vector y into equal size blocks, and have each task compute the entries of each such block. Figure 4.19 illustrates this decomposition for the case in which we decompose the computations among three different tasks. In this decomposition, even though each task (by virtue of the decomposition) knows a-priori which rows of matrix A it needs to access, it does not know which entries of vector b it requires. The reason is that the access pattern for b depends on the structure of the sparse matrix A, which can

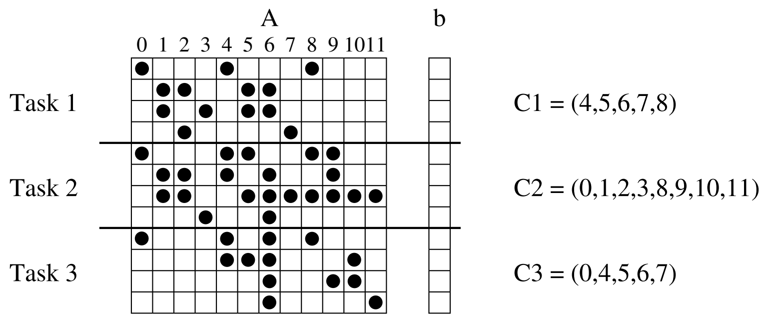


Figure 4.19 A decomposition for sparse matrix-vector multiplication. The list Ci contains the indices of b that Task i needs to access from other tasks.

be different for different matrices. A task requires a certain b_j only if it has a nonzero in column j in at least one of its rows. The irregular task interaction graph for the same problem with a different decomposition is shown in Figure 4.8(b).

Parallel computations with regular interaction patterns are often easy to program. In such parallel formulations, it is easy to ensure that the interactions at each process are also balanced. On the other hand, if the interactions are irregular, then a static mapping of tasks can lead to imbalance in interactions because some processes may perform too many interactions relative to the others.

Read-only versus Read-Write We have already learned that sharing of data among tasks leads to inter-task interaction. However, the type of sharing may affect the choice the mapping. Data sharing interactions can be categorized as either ***read-only*** or ***read-write*** interactions. As the name suggests, in read-only interactions, tasks require only a read-access to the data shared among many concurrent tasks. For example, in the decomposition for parallel matrix multiplication shown in Figure 4.1, the tasks only need to read the shared input matrices A and B. In read-write interactions, multiple tasks need to read and write on some shared data. For example, consider the problem of solving the 15-puzzle. The solution proposed in Section 4.2.3 uses an exhaustive search to find a solution. Now consider using heuristic search for solving this problem. In Section 4.2.3, each state is considered an equally suitable candidate for further expansion, In heuristic search, we use a heuristic to provide a relative approximate indication of the distance of each state from the solution (*i.e.* the potential number of moves required to reach the solution). In the case of 15-puzzle, the number of tiles that are out of place in a given state could serve as such a heuristic. The states that need to be expanded further are stored in a priority queue based on the value of this heuristic. While choosing the states to expand, we give preference to more promising states. In ths situation, the priority queue constituted shared data and tasks need both read and write access to it; they need to put the states resulting from a previous expansion into the queue and they need to pick up the next most promising state for the next expansion.

One-way versus Two-way In some interactions, the data or work needed by a task or a subset of tasks is explicitly supplied by another task or subset of tasks. Such interactions are called ***two-way*** interactions or ***producer-consumer*** interactions and usually involve predefined producer and consumer tasks. In other interactions, only one of a pair of communicating tasks initiates the interaction and completes it without interrupting the other one. Such an interaction is called a ***one-way*** interaction. All read-only interactions can be formulated as one-way interactions. Read-write interactions can be either one-way or two-way.

One-way interactions cannot be directly programmed in message passing paradigm because the source of the data to be transferred must explicitly send the data to the recipient. The shared-address-space programming paradigms can handle both one-way and two-way interactions equally easily. In message-passing paradigm, all one-way interactions must be converted to two-way interactions via program restructuring. Synchronous one-way interactions can be easily converted to two-way communications. But asynchronous (i.e., dynamic) one-way interactions can require some non-trivial program restructuring to be converted to two-way interactions. Static-regular one-way interactions are easy to convert to two-way interactions. Static-irregular one-way interactions are harder, but not impossible, to convert.

4.4 Mapping Techniques

Once a computation has been decomposed into tasks, these tasks are mapped onto processes with the objective that all tasks complete in the shortest amount of elapsed time. In order to to achieve a small execution time, the ***overheads*** of executing the tasks in parallel must be minimized. For a given decomposition, two of the key sources of overhead, namely, process idle time and inter-process interaction, are often a function of mapping. Therefore, a good mapping of tasks onto processes strives to achieve the twin objectives of (1) reducing the total amount of time some processes are idle while the others are engaged in performing some tasks, and (2) reducing the amount of time processes spend in interacting with each other due to interaction among tasks mapped onto different processes.

These two objectives often conflict with each other. For example, the objective of minimizing the interactions can be easily achieved by assigning sets of tasks that need to interact with each other onto the same process. In most cases, such a mapping will result in a highly imbalanced work-load among the processes. In fact, following this strategy to the limit will often map all tasks onto a single process. As a result, the processes with lighter load will be idle while those with a heavier load are trying to finish their tasks. Similarly, to balance the load among processes, it may be necessary to assign tasks that interact heavily to different processes. Due to the conflicts between these objectives, finding a good mapping is a non-trivial problem.

In this section, we will discuss various schemes for mapping tasks on to processes with the primary view of balancing the task workload of processes and minimizing their idle time. Reducing inter-process interaction is the topic of the later Section 4.5. The reader should be aware that assigning a balanced aggregate load of tasks to each process is a necessary but not a sufficient condition for reducing process idling. Recall that all tasks resulting from a decomposition are not all ready of execution at the same time. A task graph determines which tasks can execute in parallel and which must wait for some others to finish at a give stage in the execution of a parallel algorithm. Therefore it is possible in a certain parallel formulation that although all processes perform the same aggregate amount of work, at different times, only a fraction of the processes are active while the remainder contain tasks that must wait for other tasks to finish. Similarly, poor synchronization among interacting tasks can lead to idling if one of the tasks has to wait to send or receive data from another task. Therefore, a good mapping must ensure that the computations and interactions among processes at each stage of the execution of the parallel algorithm are well balanced.

Mapping techniques used in parallel algorithms can be broadly classified into two categories: ***static*** and ***dynamic***. Static mapping techniques distribute the tasks among processes prior to the execution of the algorithm. Dynamic mapping techniques distribute the work among processes during the execution of the algorithm.

The parallel programming paradigm and the characteristics of tasks and the interactions among them determine a static or a dynamic mapping is more suitable. Algorithms that make use of static mapping are in general easier to design and program. On the other hand algorithms that require dynamic mapping are usually more complicated, particularly in the message-passing programming paradigm.

If tasks are generated dynamically, then they must be mapped dynamically too. For statically generated tasks, either static or dynamic mapping may lead to a better parallel algorithm. The choice of a good mapping in this case depends on several factors, including the knowledge of task sizes, the size of data associated with tasks, the characteristics of inter-task interactions, and even the parallel programming paradigm. Even when task sizes are known, in general, the problem of containing an optimal mapping is an NP-complete problem for non-uniform tasks. However, for most practical cases, relatively inexpensive heuristics provide fairly acceptable approximate solutions to the optimal static mapping problem. If task sizes are unknown, then a static mapping can potentially lead to serious load-imbalances and dynamic mappings are usually more effective. Similarly, dynamic mapping may be preferable when tasks have dynamic or irregular interaction patterns, especially in the shared-address-space paradigm. If the amount of data associated with tasks is large relative to the computation, then a dynamic mapping may entail moving this data among processes. The cost of this data movement may outweight some other advantages of dynamic mapping and may render a static mapping more suitable. However, in a shared-address-space paradigm, dynamic mapping may work well even with large data associated with tasks if the interaction is read-only. The reader should be aware that the shared-address-space programming parading does not automatically provide immunity against data-movement costs. If the underlying hardware is NUMA (Chapter **??**), then the data may physically move from a distant memory. Even in a cc-UMA architecture, the data may have to move from one cache to another.

Having discussed the guidelines for choosing between static and dynamic mappings, we now describe various schemes of these two types of mappings in detail.

4.4.1 Schemes for Static Mapping

Static mapping is often, though not exclusively, used in conjunction with a decomposition based on data partitioning. Static mapping is also used for mapping certain problems that are expressed naturally by a static task graph. In the following subsections, we will discuss mapping schemes based on data partitioning and task partitioning.

Mappings Based on Data Partitioning

In this section, we will discuss mappings based on partitioning two of the most common ways of representing data in algorithms, namely, arrays and graphs.

Array Distribution Schemes In a decomposition based on partitioning data, the tasks are closely associated with portions of data by the owner-compute rule. Therefore, mapping the relevant data onto the processes is equivalent to mapping tasks onto processes. We now study some commonly used techniques of distributing arrays or matrices among processes.

Block Distributions

Block distributions are some of the simplest ways to distribute an array and assigns uniform contiguous portions of the array to different processes. In these distributions, a d-dimensional array is distributed among the processes such that each process receives a contiguous block of array entries along a specified subset of array dimensions. Block distribution of arrays are particularly suitable when there is a locality of interaction, *i.e.*, computation of an element of an array requires other nearby elements in the array.

For example, consider a 2-dimensional array A with dimension sizes $\{n_1, n_2\}$. We can now select one of these dimensions, *e.g.*, the first dimension, and partition the array into p parts such that the kth part contains elements $a_{i,j}$ where $kn_1/p \leq i < (k+1)n_1/p$ and $0 \leq j < n_2$. That is, if we think of the first dimension as corresponding to the rows of the array, then each partition contains a block of n_1/p consecutive rows of A. On the other hand, if we choose to partition along the second dimension, then each partition will contain a block of n_2/p consecutive columns of A. These row- and column-wise array distributions are illustrated in Figure 4.20.

Similarly, instead of selecting a single dimension, we can select multiple dimensions. For instance, in the case of array A we can select both dimensions and partition the matrix into blocks such that each block corresponds to a $n_1/p_1 \times n_2/p_2$ section of the matrix, with $p_1 * p_2$ being the number of processes. Figure 4.21 illustrates two different two-dimensional distributions, on a 4×4 and 2×8 process grid, respectively. In general, given a d-dimensional array, we can distribute it using up to a d-dimensional block distribution.

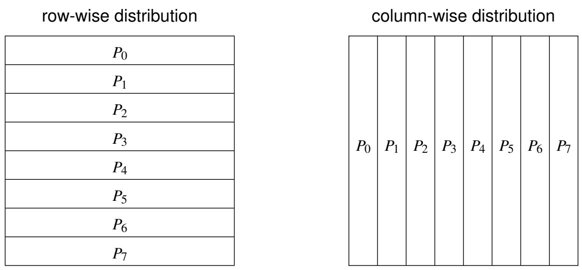


Figure 4.20 Examples of a row- and column-wise distribution of an array.

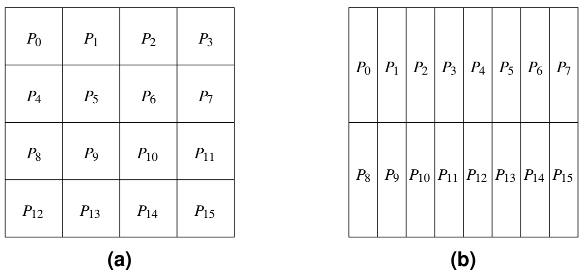


Figure 4.21 Examples of two-dimensional distributions of an array, (a) on a 4×4, and (b) on a 2×8 process grid.

Using these block distributions we can load-balance a variety of parallel computations that operate on multi-dimensional arrays. For example, consider the computation of multiplying two $n \times n$ matrices A and B to get $C = A \times B$. As discussed in Section 4.2.2, one way of decomposing this computation is to partition the output matrix C. Since, each entry of C requires the same amount of computation, we can balance the computations by distributing C using either a one- or two-dimensional block distribution. In the first case, each process will get a block of n/p rows (or columns) of C, whereas in the second case, each process will get a block of size $n/\sqrt{p} \times n/\sqrt{p}$. In either case, the process will be responsible for computing the entries of C that it is assigned to.

As this example illustrates, quite often we have the choice of mapping the computations using either a one- or a two-dimensional distribution (and even more choices in the case of higher dimensional arrays). In general, higher dimensional distributions allow us to use more processes. For example, in the case of matrix-matrix multiplication, a one-dimensional distribution will allows us to use up to n processes by assigning a single row of C to each process. On the other hand, a two-dimensional distribution will allows us to use up to n^2 processes by assigning a single element of C to each process.

In addition to allowing a higher degree of concurrency, higher dimensional distributions also sometimes help in reducing the amount of interactions among the different processes for many problems. For example consider again the matrix-matrix multiplication. In the case of a one-dimensional partitioning along the rows, each process will need to access the

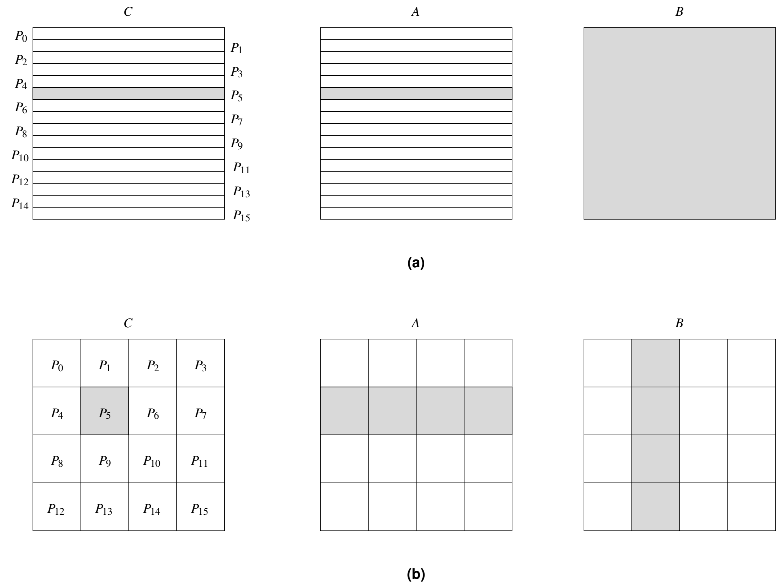


Figure 4.22 Different decompositions require different amounts of data to perform the same amount of computation in matrix multiplication.

corresponding n/p rows of matrix A and the entire matrix B, as shown in Figure 4.22(a) for process P_5. Thus the total amount of data that needs to be accessed is $n^2/p + n^2$. However, in the case of the two dimensional distribution, each process needs to access $n/\sqrt{p}$ rows of matrix A and $n/\sqrt{p}$ columns of matrix B, as shown in Figure 4.22(b) for the same process. In this case, the total amount of shared data that each process needs to access is $2n^2/\sqrt{p}$, which is significantly smaller compared to the one-dimensional case.

Cyclic and Block-Cyclic Distributions

If the amount of work differs for different elements of a matrix, a block distribution can potentially lead to load imbalances. For example, consider the decomposition for LU factorization shown in Figure 4.4 with a 3×3 two-dimensional block partitioning of the matrix. If we assign all tasks that compute a certain block to a process in 9-process ensemble, then a significant amount of idle time will result. First, computing different blocks of the matrix requires different amount of work. For example, in Figure 4.4, computing the final value of $A_{1,1}$ (which is actually $L_{1,1}U_{1,1}$) requires only one task—Task 1. On the other hand, computing the final value of $A_{3,3}$ requires three tasks—Task 9, Task 13, and Task 14. Secondly, even if all the computations on a block have not finished, the process working on that block may still idle. This idling can occur if the constraints imposed by the task graph do not allow the remaining tasks on this process to proceed until one or more tasks mapped onto other processes are completed. The ***block-cyclic distribution*** is a variation of the block distribution scheme that can be used in such cases to alleviate the

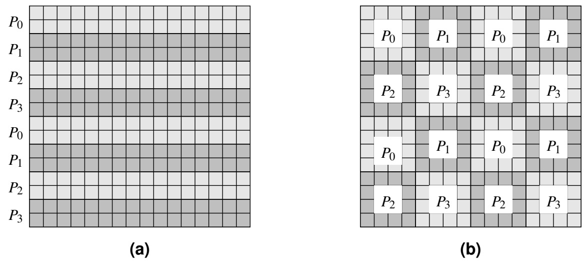


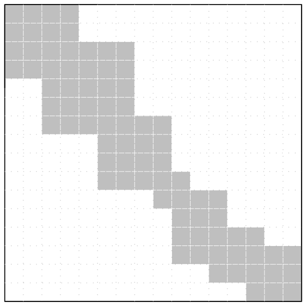
Figure 4.23 Examples of one- and two-dimensional block-cyclic distributions. (a) The rows of the array are grouped into blocks each consisting of two rows, resulting in eight blocks of rows. These blocks are distributed to four processes in a wrap-around fashion. The process storing each particular block is shown on the left of the array. (b) The matrix is blocked into 16 blocks each of size 4×4, and it is distributed among the processes in a wrap-around fashion. Each block is labeled by the process that is assigned the task of computing that block.

idling problem. A detailed description of LU factorization with block-cyclic mapping is covered in Chapter **??**.

In a one-dimensional block-cyclic distribution of a matrix among p processes, the rows (columns) of an $n \times n$ matrix are divided into αp groups of $n/(\alpha p)$ consecutive rows (columns), where $1 \leq \alpha \leq n/p$. Now, these blocks are distributed among the p processes in a wrap around fashion such that block b_i is assigned to process $P_{i\%p}$ (recall that '%' is the modulo operator). This distribution assigns α blocks of the matrix to each process, but each subsequent block that gets assigned to the same process is p blocks away. The block-cyclic distribution can be easily extended to higher dimensions. For example, we can obtain a two-dimensional block-cyclic distribution of an $n \times n$ array by partitioning it into square blocks of size $\alpha\sqrt{p} \times \alpha\sqrt{p}$ and distributing them among the processes in a wrap-around fashion. Figure 4.23 illustrates one- and two-dimensional block cyclic distributions of a two-dimensional array.

The reason that a block-cyclic distribution is able to significantly reduce the amount of idling is that all processes have a sampling of tasks from all parts of the matrix. As a result, even if different parts of the matrix require different amounts of work, the overall work on each process balances out. Also, since the tasks assigned to a process belong to different parts of the matrix, there is a good chance that at least some of them are ready for execution at any given time.

Note that if we increase α to its upper limit of n/p, then each block is a single row (column) of the matrix in a one-dimensional block-cyclic distribution and a single element of the matrix in a two-dimensional block-cyclic distribution. Such a distribution is known as a ***cyclic distribution***. A cyclic distribution is an extreme case of a block-cyclic distribution and can result in an almost perfect load balance due to the extreme fine-grained underlying decomposition. However, since a process does not have any contiguous data to work on, the resulting lack of locality may result in serious performance penalties. On



(a)

P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

(b)

Figure 4.24 Using a the block-cyclic distribution shown in (b) to distribute the computations performed in array (a) will lead to load imbalances.

the other hand, the lower limit of 1 for the value of α results in maximum locality, but the distribution degenerates to a block distribution. Therefore, an appropriate value of α must be used to strike a balance between locality preservation and load balance.

As it was the case with block-distributions, higher dimensional block-cyclic distributions are in general preferable as they may incur a lower volume of inter-task interaction.

Randomized Block Distributions

The block-cyclic distribution is unable to balance computations when the distribution of work has some special patterns. For example consider the sparse matrix shown in Figure 4.24(a) in which the shaded areas correspond to regions containing non-zero elements. If this matrix is distributed using a two-dimensional block-cyclic distribution, as illustrated in Figure 4.24(b), then we will end up assigning more non-zero blocks to processes P_0, P_5, P_{10}, and P_{15}, than on any other processes. In fact some processes, like P_{12} will not get any work.

In this case, ***randomized block distribution*** can be used, which is a more general form of the block distribution. Just like a block-cyclic distribution, load balance is sought by partitioning the array into many more blocks than the number of available processes. However, the blocks are uniformly and randomly distributed among the processes. A one-dimensional randomized block distribution can be achieved as follows. A vector V of length αp (*i.e.*, equal to the number of blocks) is used and $V[j]$ is set to j for $0 \leq j < \alpha p$. Now, V is randomly permuted, and process P_i is assigned the blocks stored in $V[i\alpha \ldots (i+1)\alpha - 1]$. A two-dimensional randomized block distribution can be computed similarly by randomly permuting two vectors and using them to choose the row and column indices of the blocks to be assigned to each process. As illustrated in Figure 4.25, the random block distribution is more effective in load balancing the computations performed in Figure 4.24.

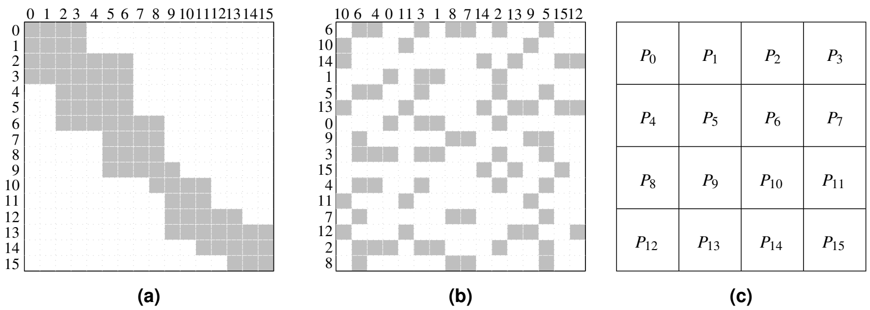


Figure 4.25 Using a two-dimensional random block distribution shown in (b) to distribute the computations performed in array (a), as shown in (c).

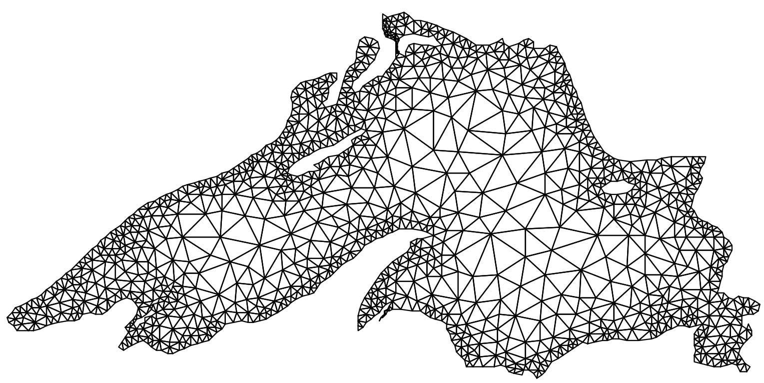


Figure 4.26 A mesh used to model lake superior.

Graph Partitioning The array-based distribution schemes that we described so far are quite effective in balancing the computations and minimizing the interactions for a wide range of algorithms that use dense matrices, and have structured and regular interaction patterns. However, there are many algorithms that operate on sparse data structures and for which the pattern of interaction among data elements is data dependent and highly irregular.

Numerical simulations of physical phenomena provide a large source of such type of computations. In these computations, the physical domain is discretized and represented by a mesh of elements. Computations are then performed at each mesh point and involve only points that are adjacent in the mesh. For example, in simulating water contamination in lake superior, one will create a mesh as shown in Figure 4.26, and use this mesh to perform the numerical simulation of contaminant dispersion.

Since in general, the amount of computation at each point is the same, so the load can be easily balanced by simply assigning the same number of mesh points to each process. However, a distribution of the mesh points to processes may lead to high interaction over-

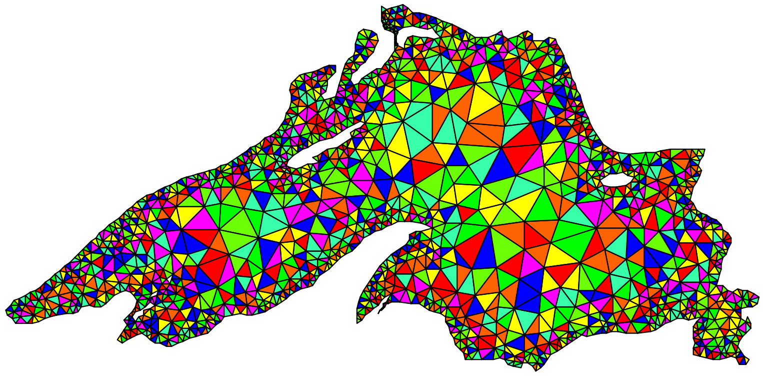


Figure 4.27 A random distribution of the mesh elements to eight processes.

heads due to excessive data sharing because the distribution may not strive to keep nearby mesh points together. For example, if each process receives a random set of points as illustrated in Figure 4.27, then each process will need to access a large set of points located at other processes to complete computations for its assigned portion of the mesh.

Ideally, we would like to distribute the mesh points in a way that balances the load and at the same time minimized the amount of data that each process needs to access in order to complete its computations. We can achieve this by using ***graph partitioning***. Specifically, a graph $G = (V, E)$ is formed such that each node $u \in V$ corresponds to computation and an edge $e \in E$ between two vertices corresponds to dependencies. That is, an edge (u, v) indicates that in order to compute u we need some information that is stored in node v. Since, each node needs information from few other nodes, G is sparse. Next, a graph partitioning algorithm is used to partition G into p partitions, such that all partitions have the roughly same number of nodes, and the number of edges that straddle partitions is minimized. Finally, each one of these p partitions is assigned to one process. For example, in the case of mesh computations, a graph corresponding to the mesh is formed and a partitioning algorithm is used to distribute this graph (*i.e.*, mesh), as shown in Figure 4.28. As a result, each process is assigned a contiguous region of the mesh such that the total number of mesh points that needs to be accesses is minimized.

Mappings Based on Task Partitioning

A mapping based on partitioning a task graph and mapping its nodes onto processes can be used when the computation is naturally expressible in the form of a static task graph with tasks of known sizes. As usual, this mapping must seek to achieve the often conflicting objectives of minimizing idle time and minimizing the interaction time of the parallel algorithm. Determining an optimal mapping for a general task graph is an NP-complete problem. However, specific situations often lend themselves to a simpler optimal or acceptably approximate solution.

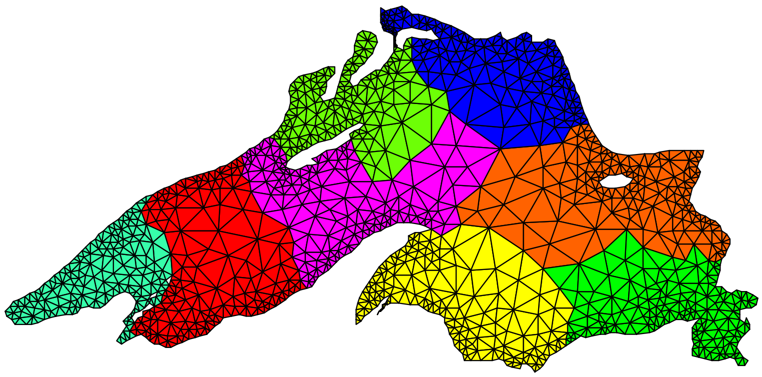


Figure 4.28 A distribution of the mesh elements to eight processes, by using a graph-partitioning algorithm.

As a simple example of a task-graph mapping, consider a task graph that is a perfect binary tree. Such a task graph can occur in practical problems with recursive decomposition, such as the decomposition for finding the minimum of a list of number (Figure 4.10). Figure 4.29 shows a mapping of this task graph onto processes that are running on a hypercube interconnection network. Notice how the nodes of the tree are labeled starting from 000 for the root and changing one bit at each level starting from the most significant to the least significant. It is easy to see that this mapping minimizes the interaction overhead by mapping many interdependent tasks onto the same process (*i.e.*, the tasks along a straight branch of the tree) and others on processes only one communication link away from each other. Even on an architecture that provides the same level of connectivity among all nodes and the number of links between interacting processes is not a concern, this is a good mapping. Although there is some inevitable idling (*e.g.*, when process 000 works on the root task, all other processes are idle), this idling is inherent in the task graph. The mapping shown in Figure 4.29(b) does not induce any further idling and all tasks that are permitted to be concurrently active by the task graph are mapped onto different processes for parallel execution.

For some problems, an approximate solution to the problem of finding a good mapping can be obtained by partitioning the task interaction graph described in Section 4.1.2 in the context of the problem of sparse matrix-vector multiplication. A good mapping for p processes can be obtained by partitioning a task graph into p nearly parts of almost equal weight such that such of weights of the edges crossing partition boundaries is minimized. Note that if all nodes and edges have unit weights, then the graph partitioning problem is similar to the one discussed in the context of partitioning a graph data structure. In both weighted and unweighted forms, this is an NP-complete problem; however, good approximate algorithms are available. Figure 4.30 shows such a partitioning for the task interaction graph of the sparse matrix vector multiplication problem shown in Figure 4.8

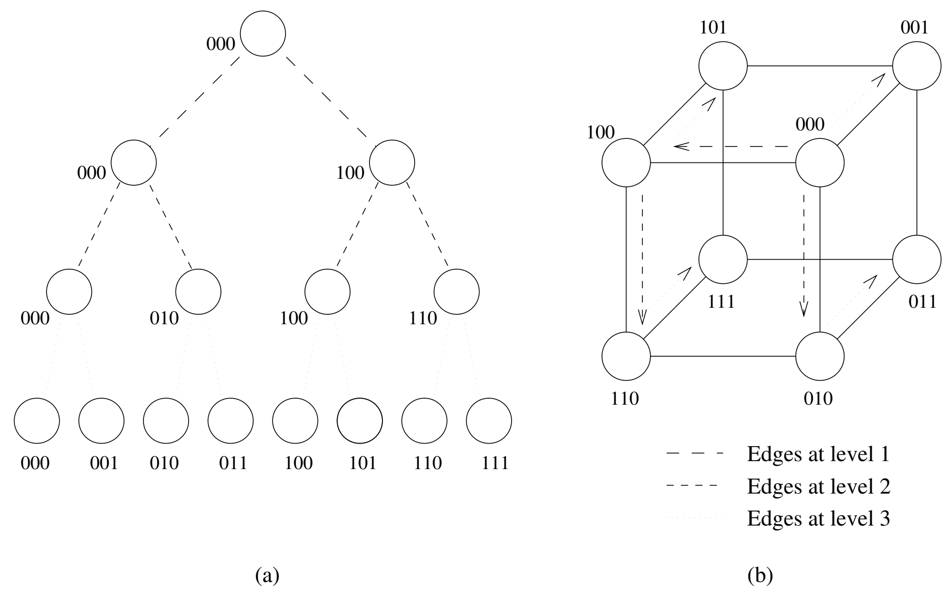


Figure 4.29 Mapping of a binary tree task graph onto a hypercube of processes.

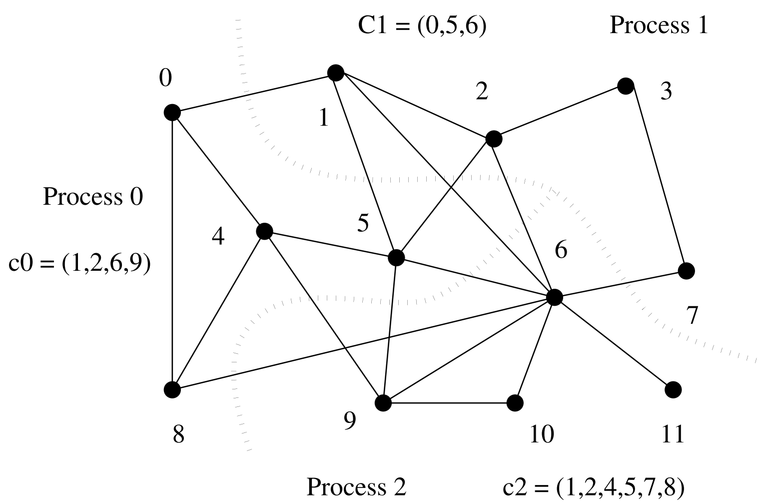


Figure 4.30 Reducing interaction overhead in sparse matrix-vector multiplication by partitioning the task interaction graph.

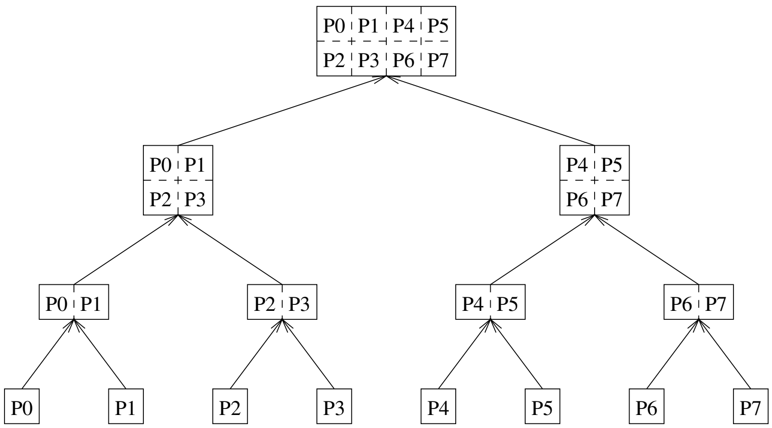


Figure 4.31 An example of hierarchical mapping of a task graph. Each node represented by an array is a supertask. The partitioning of the arrays represents subtasks, which are mapped onto eight processes.

for three processes. The list Ci contains the indices of b that the tasks on Process i need to access from tasks mapped onto other processes. For the same problem, a more naive decomposition and mapping map three tasks of the same size onto three processes, where each task computes four consecutive entries of b. Such a decomposition is shown in Figure 4.19. A quick comparison of the lists $C0$, $C1$, and $C2$ in the two cases readily reveals that the mapping based on partitioning the task interaction graph entails fewer exchanges of elements of b between processes than the naive decomposition and mapping.

Hierarchical Mappings

Certain algorithms are naturally expressed as task graphs; however, a mapping based solely on the task graph may suffer from load-imbalance or inadequate concurrency. For example, in the binary-tree task graph of Figure 4.29, only a few tasks are available for concurrent execution in the top part of the tree. If the tasks are big enough, then a better mapping can be obtained by a further decomposition of the tasks into smaller subtasks. In the case where the task graph is a binary tree with four levels, the root task can be partitioned among eight processes, the tasks at the next level can be partitioned among four processes each, followed by tasks partitioned among two processes each at the next level. The eight leaf tasks can be have a one-to-one mapping onto the processes. Figure 4.31 illustrates such hierarchical mapping.

An important practical problem to which the hierarchical mapping example discussed above applies directly is that of sparse matrix factorization, which is discussed in detail in Chapter **??**. The high-level computations in sparse matrix factorization are guided by a task graph which is known as an *elimination graph* (*elimination tree* if the matrix is symmetric). However, the tasks in the elimination graph (especially the ones closer to the root) usually involve substantial computations and are further decomposed into subtasks using data decomposition. A hierarchical mapping, using task partitioning at the top layer and array partitioning at the bottom layer is then applied to this hybrid decomposition. In general, a hierarchical mapping can have many layers and any of the decomposition and mapping techniques may be suitable for each layer.

4.4.2 Schemes for Dynamic Mapping

Dynamic mapping is necessary in situations where a static mapping may result in a highly imbalanced distribution of work among processes. Since the primary reason for using a dynamic mapping is balancing the work-load among processes, dynamic mapping is often referred to as dynamic load-balancing. Schemes for dynamic mapping or load balancing can be either ***centralized*** or ***distributed***.

Centralized Schemes

In a centralized dynamic load balancing scheme, all executable tasks are maintained in a common centralized data structure or by a special process or a subset of processes. If a special process is designated to manage the pool of available tasks, then it is often referred to as the ***master*** and the other processes that depend on the master to obtain work are referred to as ***slaves***. Whenever a process has no work, it takes a portion of available work from this data structure or process. Whenever a new task is generated, it is added to this centralized data structure or reported to the master process. Centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability. As more and more processes are used, the large number of accesses to the common data structure or the master process may become a bottleneck.

As an example of a scenario where centralized mapping may be useful, consider the problem of sorting the entries in each row of an $n \times n$ matrix A. Serially, this can be accomplished by the following simple program segment:

```
for (i=1; i<n; i++)
  sort(A[i],n);
```

Recall that the time to sort an array using some of the popular sorting algorithms can vary significantly depending on the initial order of the elements to be sorted. Therefore, each iteration of the loop in the program shown above will take different amounts of time and a naive mapping of the task of sorting an equal number of rows to each process may lead to a load-imbalance. A possible solution to the potential load-imbalance problem in this case would be to maintain a central pool of indices of the rows that have yet to be sorted. Each process picks up an available index, deletes it, and sorts the row with that index whenever it is idle, as long as the pool of indices is not empty. This methods of scheduling the independent iterations of a loop among parallel processes is known as ***self scheduling***.

The assignment of a single task to a process at a time, is quite effective in balancing the computation; however, it may lead to bottlenecks in accessing the shared work queue, especially if each task (*i.e.*, each loop iteration in this case) does not require a large enough amount of computation. If the average size of each task is M, and it takes Δ time to assign work to a process, then at most M/Δ processes can be kept busy effectively. The bottleneck can be eased by getting more than one task at a time. In ***chunk scheduling***, every time a process runs out of work it gets a group of tasks. The potential problem with such a scheme is that it may lead to load-imbalances if the number of tasks (*i.e.*, chunk) assigned in a single step is large. The danger of load-imbalance due to large chunk sizes can be reduced by decreasing the chunk-size as the program executes. That is, initially the chunk size is large, but as the number of iterations left to be executed decreases, the chunk size also decreases. A variety of schemes have been developed for gradually adjusting the chunk size, that decrease the chunk size either linearly or non-linearly.

Distributed Schemes

In a distributed dynamic load balancing scheme, each process can send work to or receive work from any other process. These methods do not suffer from the bottleneck associated with the centralized schemes. Some of the critical parameters of a distributed load balancing scheme are as follows.

- How the sending and receiving processes are paired together.
- Is the work transfer initiated by sender (sender initiated load balancing) or initiated by receiver (receiver initiated load balancing).
- How much work is transferred in each exchange? If too little work is transferred, then the receiver may not have receive enough work and frequent transfers resulting in excessive interaction may be required. If too much work is transferred, then the sender may be out of work soon, again resulting in frequent transfers.
- When is the work transfer performed? For example, in receiver initiated load balancing, work may be requested when the process has actually run out of work or when the receiver has too little work left and anticipates being out of work soon.

A detailed study of each of these parameters is beyond the scope of this chapter. These load balancing schemes will be revisited in the context of parallel algorithms to which they apply when we discuss these algorithms in the later chapters. In particular, Chapter **??** in the context of parallel search algorithms.

Ananth, what is the label for the search chapter?

Suitability To Parallel Architectures

Note that in principle, both centralized and distributed mapping schemes can be implemented in both message passing and shared-address-space paradigms. However, by its very nature any dynamic load balancing scheme requires movement of tasks from one process to another. Hence, for such schemes to be effective on message passing computers, the size of the tasks in terms of computation should be much higher than the size of the data associated with the tasks. In a shared-address-space paradigm, the tasks don't need to be moved explicitly, although there is some implied movement of data to local caches or memory banks of processes. In general, the computational granularity of tasks to be moved can be much smaller on shared-address architecture than on message passing architectures.

4.5 Methods for Containing Interaction Overheads

As noted earlier, reducing the interaction overhead among concurrent tasks is important for an efficient parallel program. The overhead that a parallel program incurs due to interaction among its processes depends on three main factors:

(i) The volume of data exchanged during interactions.

(ii) The frequency of interaction.

(iii) Spatial and temporal pattern of interactions.

Ananth, pls provide label for sectipn where message startup cost is discussed

The volume or the total amount of information shared or exchanged between processes has a direct bearing on the interaction overhead of a parallel program. As discussed in Section **??**, on most parallel architectures, each interaction operation has a high startup cost and a relatively lower cost for each unit of the information exchanged. Therefore, for the same total volume and pattern of interaction, several small interactions are costlier than fewer large interactions. The spatial and temporal pattern of interaction determines contention for shared resources and the balance and overlap among interactions and computations.

In this section, we will discuss some general techniques that can be used to reduce the interaction overheads incurred by parallel programs. These techniques manipulate one or more of the three factors above in order to reduce the interaction overhead. Some of these are applicable while devising the decomposition and mapping schemes for the algorithms and some are applicable while programming the algorithm in a given paradigm. All techniques may not be applicable in all parallel programming paradigms and some of them may require support from the underlying hardware.

4.5.1 Maximizing Data Locality

In most non-trivial parallel programs, the tasks executed by different processes require access to some common data. For example, in sparse matrix-vector multiplication $y = Ab$, in which tasks correspond to computing individual elements of vector y (Figure 4.8), all elements of the input vector b need to be accessed by more than one tasks. In addition to sharing the original input data, in many computations, some processes require data that may have been generated by other processes. This sharing of data often leads to interaction overheads in a parallel program. The interaction overheads can be reduced by using techniques that promote the use of local data or data that has been recently fetched.

Data locality enhancing techniques encompass a wide range of schemes that try to minimize the volume of non-local data that is accessed, maximize the reuse of recently accessed data, and minimize the frequency of accesses. In many cases, these schemes are similar in nature to the data reuse optimizations often performed in modern cache based microprocessors.

Minimize Volume of Data-Exchange A fundamental technique for reducing the interaction overhead is to try to minimize the overall volume of shared data that needs to be accessed by the concurrent processes. This is akin to maximizing the temporal data locality; i.e., making as many of the consecutive references to the same data as possible. Clearly, performing as much of the computation as possible using locally available data obviates the need of bringing in more data into local memory or cache for a process to perform its tasks. As discussed previously, one way of achieving this is by using appropriate decomposition and mapping schemes. For example, in the case of matrix multiplication, we saw that by using a two-dimensional mapping of the computations to the processes we were able to reduce the amount of shared data (*i.e.*, matrices A and B) that needs to be accessed by each task to $2n^2/\sqrt{p}$ as opposed to $n^2/p + n^2$ required by a one-dimensional mapping (Section 4.4.1). In general, using higher dimensional distribution often helps in reducing the volume of nonlocal data that needs to be accessed.

Another way of decreasing the relative amount of shared data that needs to be accessed is to increase the amount of work performed by each task by using fewer tasks. This approach essentially builds data locality by simply assigning more computations to each task. Note that this approach is in direct conflict with the objective of using a large number of tasks to exploit as much concurrency as possible and to help with load-balancing.

We can also decrease the amount of shared data that needs to be accessed by using local data to store intermediate results, and perform the shared data access to only place the final results of the computation. For example, consider computing the dot product of two vectors in parallel such that each task multiplies certain pairs of elements. Rather than adding each individual product of a pair of numbers to the final result, each task can first create a partial dot product in its own local location, and only access the final shared location once to add its partial result.

Minimize Frequency of Interactions Minimizing interaction frequency is important in reducing the interaction overheads in parallel programs because there is a relatively high startup cost associated with each interaction on most practical architectures. Interaction frequency can be reduced by restructuring the algorithm such that shared data that needs to be accessed are accessed in large pieces. Thus, by amortizing the startup cost over large accesses, we can reduce the overall interaction overhead, even if such restructuring does not necessarily reduce the overall volume of shared data that needs to be accessed. This is akin to increasing the spatial locality of data access; i.e., ensuring the proximity of consecutively accessed data locations. On a shared-address space architecture, each time a word is accessed, an entire cache line containing many words is fetched. If the program is structured to have spatial locality, then fewer cache lines are accessed. On a message-passing system, spatial locality leads to fewer message-transfers over the network because each message can transfer larger amounts of useful data. The number of messages can sometimes be reduced further on a message-passing system by combining messages between the same source-destination pair into larger messages if the interaction pattern permits and if the data for multiple messages is available at the same time, albeit in separate data structures.

4.5.2 Minimizing Contention and Hot Spots

Our discussion so far has been largely focused on reducing interaction overheads by directly or indirectly reducing the frequency and volume of data transfers. However, the data-access and inter-task interactions patterns can often lead to contention that can increase the overall interaction overhead. In general, contention occurs when multiple tasks try to concurrently access the same resources. For example, multiple simultaneous transmissions of data over the same interconnection link, multiple simultaneous accesses to the same memory block, or multiple processes sending messages to the same process at the same time, quite often lead to contention as only one of the multiple operations can proceed at a time.

One way of reducing contention is to redesign the parallel algorithm to access data in contention free patterns. For example, consider the problem of multiplying two matrices $C = AB$, using the two-dimensional partitioning shown in Figure 4.22(b). Let p be the number of tasks with a one to one mapping of tasks onto processes. Let each tasb be responsible for computing a unique $C_{i,j}$, for $0 \leq i, j < \sqrt{p}$. The straight forward way of performing this computation is for $C_{i,j}$ to be computed according to the following formula (written in matrix-block notation):

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} * B_{k,j} \tag{4.1}$$

Looking at the memory access patterns of the above equation, we see that at any one of the $\sqrt{p}$ steps, $\sqrt{p}$ tasks will be accessing the same block of A and B. In particular, all the tasks that work on the same row of C will be accessing the same block of A. Similarly, all the tasks working on the same column of C will be accessing the same block of B. The need to concurrently access these blocks of matrices A and B will create contention on both NUMA shared-address space and message-passing parallel architectures. One way of eliminating this contention is to modify the order in which the block multiplications are performed in Equation 4.1. A contention-free way of performing these block-multiplications is to compute $C_{i,j}$ by using the formula

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} * B_{(i+j+k)\%\sqrt{p},j}, \tag{4.2}$$

where '%' denotes the modulo operation. By using this formula, all the tasks $P_{*,j}$ that work on the same row of C will be accessing block $A_{*,(*+j+k\%\sqrt{p}}$, which is different for each task. Similarly, all the tasks $P_{i,*}$ that work on the same column of C will be accessing block $B_{(i+*+k)\%\sqrt{p},*}$, which is also different for each task. Thus, by simply re-arranging the order that the block-multiplications are performed, we can completely eliminate the contention.

In certain parallel algorithms requiring dynamic mapping, the contention can be reduced by choosing a distributed mapping scheme over a centralized one, even though the former may be harder to implement.

4.5.3 Overlapping Computations with Interactions

The amount of time that processes spend waiting for shared data to arrive or to receive additional work after an interaction has been initiated can be reduced or even mostly eliminated by doing some useful computations during this waiting time. There are a number of techniques that can be used to overlap computations with interactions.

A simple way of overlapping is to initiate an interaction early enough so that it is completed before it is needed for computation. To achieve this, we must be able to identify computations that can be performed before the interaction and do not dependent on it. Then the parallel program must be structured to initiate the interaction at an earlier point in the execution than it is needed in the original algorithm. Typically, this is possible if the interaction pattern is spatially and temporally static (and therefore, predictable) or if multiple tasks that are ready for execution are available on the same process so that if one blocks to wait for an interaction to complete, the process can work on another task. The reader should note that by increasing the number of parallel tasks to promote computation-interaction overlap, we are reducing the granularity of the tasks, which, in general tends to increase overheads. Therefore, this technique must be used judiciously.

In certain dynamic mapping schemes, as soon as a process runs out of work, it requests and gets additional work from another process. It then waits for the request to be serviced. If the process can anticipate that it is going to run out of work and initiate a work transfer interaction in advance, then it may continue towards finishing the tasks at hand while the request for more work is being serviced. Depending on the problem, estimating the amount of remaining work, may be easy or hard.

In most cases, overlapping computations with interaction requires support from the programming paradigm, operating system, and the hardware. The programming paradigm must provide a mechanism to allow interactions and computations to proceed concurrently. This mechanism should be supported by the underlying hardware. Disjoint address-space paradigms and architectures usually provide this support via non-blocking message passing primitives. In this mechanism, there are functions that can be used to send and receive messages that return control to the user's program before they have actually completed. Thus, the program can use these primitives to initiate the interactions, and then proceed with the computations. If the hardware permits computation to proceed concurrently with message transfers, then the interaction overhead can be reduced significantly.

On an shared-address space architecture, the overlapping of computations and interaction can also be assisted by prefetching hardware. In this case, an access to shared data is nothing more than a regular load and store instruction, and it is executed because it is needed for the computations that are currently being performed. The prefetch hardware can anticipate the memory addresses that will need to be accessed in the immediate future, and can initiate the access in advance of when they are needed. In the absence of prefetching hardware, the same effect can be achieved by a compiler that detects the access pattern and places pseudo-references to certain key memory locations before these locations are actually utilized by the computation. The degree of success of this scheme is dependent upon the available structure in the program that can be inferred by the prefetch hardware and by the degree of independence with which the prefetch hardware can function while computation is in progress. In addition, the path along the memory hierarchy that prefetching hardware indents to use must be free at the time when prefetching is initiated. This is a function of several factors including memory latency and bandwidth relative to the CPU speed and the number of memory references in the program relative to the amount of computation performed involving the referenced data.

ananth, pls provide label for the TERA reference

Another possible overlapping technique applicable on multiprogramming kernel and multithreaded microprocessors (such as TERA **??**) is to further break up the computations of each task into different concurrent streams or threads that are scheduled on the same processor. Whenever an executing thread starts waiting for an interaction to finish, it is taken off the processor, and a different thread is scheduled. This scheme has the advantage that it does not require the programmer to explicitly restructure the computations in order to hide the interaction cost. However, this comes at the expense of having to decompose the computation into fine-grained tasks (*i.e.*, derive additional concurrency) which may increase overheads. Also, these multiple threads may fragment the cache in some cases; *i.e.*, destroy spatial locality of data reference in the program.

4.5.4 Replicating Data and Computations

Replication of data or computations is another technique that may be useful in reducing interaction overheads.

In some parallel programs, multiple processes may require frequent read-only access to shared data-structure, such as a hash-table, in an irregular pattern. Unless the additional memory requirements are prohibitive, it may be best in a situation like this to replicate a copy of the shared data structure on each process so that after the initial interaction during replication, all subsequent accesses to this data-structure are free of any interaction overhead. Recall from Section 4.2.2 the algorithm that computes the frequency of a set of itemsets I in a transaction database T. In such problems, usually, a lookup structure similar to a hash table is built to efficiently determine the existence of an itemset in the set of transactions. As discussed earlier, one way of decomposing this computation is to partition the transactions, and let each task compute the frequency of the itemsets based solely on its portion of the database. For every transaction that each task processes, it needs to determine whether or not it contains any itemsets from I. Therefore, every task needs to access the shared lookup-structure very frequently and this access is done in a read-only fashion. Depending on the underlying architecture and the programming paradigm, accessing this shared data structure may become a bottleneck and incur a large overhead. One way of significantly reducing this overhead is to replicate this shared data structure among all the processes. Thus, we essentially convert shared data into data that are local to each process; thus, eliminating the data sharing overhead.

Explicit data replication is particularly suited for architectures and programming paradigms in which read-only access to shared data is significantly more expensive or harder to express than local data accesses. Therefore, disjoint address-space architectures and programming paradigms benefit the most from data replication because it may reduce interaction overhead and and also significantly simplify the writing of the parallel program.

Even though data replication, in general, does not help in reducing the programming complexity of shared address-space programming paradigms, it may still be beneficial in reducing the interaction overhead. In the case of NUMA architectures, replicating shared data will reduce the interaction overhead for two reasons. First it brings the frequently accessed data closer to the task, thus reducing latency and transfer time, and second, it eliminates contention, since the data are being accessed by only a single task. This second reason is beneficial for reducing interaction overheads on a UMA architecture, as well. If there is a single copy of the frequently accessed data, then the likelihood that multiple tasks will try to access the same memory bank increases, causing these accesses to get serialized. However, if the data are replicated, then memory bank access contention will be significantly reduced, provided that the programming paradigm and underlying hardware allow us to explicitly place data in the memory banks. Also note that in order for data replication to help in reducing interaction overheads on cache-based shared address-space architectures, the amount of data that is frequently accessed should be significantly larger than the size of the cache at each processor. This is because, the cache serves as an automatic data replication mechanism. For problems in which the working sets are small, the cache may automatically improve performance.

Data replication however, does not come without its own cost. Data replication increases the memory requirements of a parallel program. The aggregate amount of memory required to store the replicated data increases linearly with the number of concurrent processes. This may limit our ability to solve larger problems (with inherently larger data sets) by using a larger parallel computer that can support more processes and memory if the extra memory is consumed by the replicated data. For this reason, data replication must be used selectively to replicate relatively small amounts of data.

In addition to input data, the processes in a parallel program often share intermediate results. In some situations, it may be more cost-effective for a process to compute these intermediate results than to get them from another process that generates them. In such situations, interaction overhead can be lowered by replication of some computation. For example, while performing the Fast Fourier Transform (more details in Section **??**), on an N-point series, N distinct powers of ω or twiddle factors are computed and used at various points in the computation. In a parallel implementation of FFT, different processes require overlapping subsets of these N twiddle factors. In a message-passing paradigm, it is best for each process to locally compute all the twiddle factors it needs. Although the total number of twiddle factor computations across all processes may exceed $\Theta(N)$ in the parallel implementation, it may still be faster than sharing the twiddle factors.

4.5.5 Using Optimized Collective Interaction Operations

As discussed in Section 4.3.2, often, the interaction patterns among concurrent activities are static and regular. A class of such static and regular interaction patterns are those that are performed by groups of tasks, and they are used to achieve regular data accessed or to perform certain type of computations on distributed data. A number of key such *collective* interaction operations have been identified that appear frequently in many parallel algorithms. Broadcasting some data to all the processes and adding-up numbers each residing on a different process are examples of such collective operations. The collective data-sharing operations can be classified into three categories. The first category contains operations that are used by the tasks to access data, the second category of operations are used to perform some communication-intensive computations, and finally, the third category is used for synchronization.

Highly optimized implementations of these collective operations have been developed that minimize the overheads due to data transfer as well as contention. Chapter **??** describes algorithms for implementing some of the commonly used collective interaction operations. Optimized implementations of these operations are available in library form from the vendors of most parallel computers; *e.g.*, MPI (Chapter **??**). As a result, the algorithm designer does not need to think about how these operations are implemented and needs to focus only on the functionality achieved by these operations. However, as discussed in Section 4.5.6, sometimes the interaction pattern may make it worthwhile for the parallel programmer to implement one's own collective communication procedure.

4.5.6 Overlapping Interactions with Other Interactions

Consider a parallel program in which each process performs some useful computation and spends some time interacting with other processes. Assume that the amount of computation that each process must perform is exactly the same. Although the parallel program appears to be perfectly load-balanced, that may not be the case in practice. The reason is that the pattern of interaction can introduce idling overheads if pairs of interacting processes are not ready to interact at the same time or if the volume and frequency of interaction is different on different processes. The overall overhead due to such idling is the product of the maximum overhead on any process and the total number of processes. Therefore, it may sometimes be worthwhile to reduce the maximum interaction overhead even at the cost of increasing the total volume of interaction, especially if a significant portion of interaction can be overlapped.

As an example, consider the commonly used collective communication operation of one-to-all broadcast in a message-passing paradigm with four processes P_0, P_1, P_2, and P_3. A commonly used algorithm to broadcast some data from P_0 to all other processes works as follows. In the first step, P_0 sends the data to P_2. In the second step, P_0 sends the data to P_1, and concurrently, P_2 sends the same data that it had received from P_0 to P_3. The entire operation is thus complete in two steps because the two interactions of the second step require only one time step. On the contrary, a naive broadcast algorithm would send the data from P_0 to P_1 to P_2 to P_3, thereby consuming three steps.

Now assume that a parallel segment broadcasts ten data structures one after the other. The entire interaction would require 20 steps using the first two-step broadcast algorithm. However, using the naive algorithm accomplishes the interaction in only 12 steps. In the first step, P_0 sends the first message to P_1. In the second step P_0 sends the second message to P_1 while P_1 simultaneously sends the first message to P_2. In the third step, P_0 sends the third message to P_1, P_1 sends the second message to P_2, and P_2 sends the first message to P_3. Proceeding similarly in a pipelined fashion, the last of the ten messages will be out of P_0 after ten steps and will reach P_3 in twelve. Since this method is rather expensive for a single broadcast operation, it is unlikely to be included in a collective communication library. However, the programmer must infer from the interaction pattern of the program that in this scenario, it is better to make an exception to the suggestion of Section 4.5.5.

4.6 Parallel Algorithm Models

Having discussed the techniques for decomposition, mapping, and minizing 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.

4.6.1 The Data Parallel Model

The *data parallel model* is one of the simplest algorithm models used commonly for implementing many scientific and numerical algorithms. 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, and hence an equal amount of work on different data, the decomposition of the problem into tasks is usually based on data decomposition. 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 better handle on locality. Sometimes, shared-address space can ease the programming effort, especially if the distribution of data is different in each phase.

Due to a static mapping and a regular decomposition, load balancing is fairly straightforward. Interaction overheads are 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 algorithms is dense matrix multiplication described in Section 4.1.1. In the decomposition shown in Figure 4.1, all tasks are identical; they are applied to different data.

4.6.2 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 processes. 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 the could be present 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 **??**) 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 mork.

Ananth, pls provide label of the termination-detection section from the Parallel Search chapter.

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 trade-off between load-imbalance and the overhead of accessing the work pool for adding and extracting tasks.

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.

4.6.3 The Task Graph Model

As discussed in Section 4.1, the computations in any parallel algorithm can be viewed has a task graph. The task graph may either be trivial, as in the case of matrix multiplication or non-trivial. However, in certain parallel algorithms, the task 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. Therefore, the lack of a specific mapping of tasks onto processes may make the interaction costs prohibitive in the work-pool model for such problems. 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 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. In the disjoint-address-space paradigm, transfer of work to another process will also require sending the data structures associated with work.

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

George, I need label for the old-book Section 6.4.1

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

4.6.4 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 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 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 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 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 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.

4.6.5 The Pipeline 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***. Except for 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 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 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 **??**.

4.6.6 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 either be composed 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 models. For instance, data may flow in a pipelined manner in a pattern guided by a task graph. In another scenario, the major computation may be described by a task graph, but each node of the graph may represent a supertask comprising of multiple tasks that may be suitable for data-parallel or pipelined parallelism. The sparse Cholesky factorization algorithm described in Section **??** is an example of a hybrid between the task graph and the pipelined models. The supertasks are mapped onto groups of processes based on a task graph (called elimination-tree in the context of sparse Cholesky factorization). Each supertask is decomposed into smaller tasks, on which subprocesses work in a pipelined manner.

Problems

4.1 Draw the task graphs corresponding to the decompositions for matrix multiplication shown in Figures 4.1 and 4.2.
Hint: Note that task graphs can be disconnected. Also, the edge-set of a task graph can be empty.

4.2 What are the degrees of concurrency and critical-path lengths of task graphs corresponding to the decompositions for matrix multiplication shown in Figures 4.1 and 4.2.

4.3 Prove that block steps 1 through 14 in Figure 4.4 with block size b (*i.e.*, each $A_{i,j}$, $L_{i,j}$, and $U_{i,j}$ is a $b \times b$ submatrix) are mathematically equivalent to running the algorithm of Program 4.1 on an $n \times n$ matrix A, where $n = 3b$.
Hint: Using induction on b is one possible approach.

4.4 Let d be the maximum degree of concurrency in a task graph with t tasks and a critical-path length l. Prove that $\lceil \frac{t}{l} \rceil \leq d \leq t - l + 1$.

4.5 Figure 4.4 the decomposition into 14 tasks of LU factorization of a matrix split into blocks using a 3×3 two-dimensional partitioning. If an $m \times m$ partioning is used, derive an expression for the number of tasks $t(m)$ as a function of m in a decomposition along similar lines.
Hint: Show that $t(m) = t(m-1) + m^2$.

4.6 In the context of Problem 4.5, derive an expression for the maximum degree of concurrency $d(m)$ as a function of m.

4.7 In the context of Problem 4.5, derive an expression for the critical-path length $l(m)$ as a function of m.

4.8 In the algorithm shown in Program 4.2, assume a decomposition such that each execution of Line 8 is a task. Draw a task graph and a task interaction graph.

4.9 In Program 4.2, if $n = 16$, devise a good mapping for 16 processes.

4.10 In Program 4.2, if $n = 16$, devise a good mapping for 8 processes.

4.11 Repeat Problems 4.8, 4.9, and 4.10 if the statement of Line 3 in Program 4.2 is changed to $m = (\log_2 n) - 1$.

4.12 Consider a simplified version of bucket-sort. You are given an array A. of n random integers in the range $[1 \ldots r]$ as input. The output data is r buckets, such

```
procedure FFT_like_pattern(A, n)
begin
    m := log_2 n;
    for j := 0 to m - 1 do
    begin
        k := 2^j;
        for i := 0 to n - 1 do
            A[i] := A[i] + A[i XOR 2^j];
    endfor
end FFT_like_pattern
```

Algorithm 4.2 A sample serial program to be parallelized.

that at the end of the program, Bucket i contains indices of all the elements in A that are equal to i.

- (a) Describe a decomposition based on partitioning the input data (*i.e.*, the array A) and an appropriate mapping onto p processes. Describe briefly how the resulting parallel algorithm would work.
- (a) Describe a decomposition based on partitioning the output data (*i.e.*, the set of r buckets) and an appropriate mappining onto p processes. Describe briefly how the resulting parallel algorithm would work.

4.13 In the context of Problem 4.12, which of the two decompositions leads to a better parallel algorithm? Should the relative values of n and r have a bearing on the selection of one of the two decomposition schemes?

4.14 In Section 4.2.2, the ray-tracing problem is described. Assume that the pixels whose color values are to be determined are arranged in an $m \times n$ array. Which of the array distribution schemes discussed in Section 4.4.1 is the most appropriate and why?